UNIFYING DATABASES AND INTERNET-SCALE PUBLISH/SUBSCRIBE

by

Badrish Chandramouli

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jun Yang, Supervisor

_____
Shivnath Babu

_____
Jeff Chase

_____
Carla Ellis

_____
Hui Lei

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2008

ABSTRACT

UNIFYING DATABASES AND INTERNET-SCALE PUBLISH/SUBSCRIBE

by

Badrish Chandramouli

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jun Yang, Supervisor

_____
Shivnath Babu

_____
Jeff Chase

_____
Carla Ellis

_____
Hui Lei

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2008

Abstract

With the advent of Web 2.0 and the Digital Age, we are witnessing an unprecedented increase in the amount of information collected, and in the number of users interested in different types of information. This growth means that traditional techniques, where users poll data sources for information of interest, are no longer sufficient. Polling too frequently does not scale, while polling less often may result in users missing important updates. The alternative push technology has long been the goal of *publish/subscribe systems*, which proactively push updates (*events*) to users with matching interests (expressed as *subscriptions*). The push model is better suited for ensuring scalability and timely delivery of updates, important in many application domains: personal (e.g., RSS feeds, online auctions), financial (e.g., portfolio monitoring), security (e.g., reporting network anomalies), etc.

Early publish/subscribe systems were based on predefined subjects (*channels*), and were too coarse-grained to meet the specific interests of different subscribers. The second generation of content-based publish/subscribe systems offer greater flexibility by supporting subscriptions defined as predicates over message contents. However, subscriptions are still stateless filters over individual messages, so they cannot express queries across different messages or over the event history. The few systems that support more powerful database-style subscriptions do not address the problem of efficiently delivering updates to a large number of subscribers over a wide-area network. Thus, there is a need to develop next-generation publish/subscribe systems that unify the support for richer database-style subscription queries and flexible wide-area notification. This support needs to be complemented with robust processing and dissemination techniques that scale to high event rates and large databases, as well as to a large number of subscribers over the Internet.

The main contribution of our work is a collection of techniques to support efficient

and scalable event processing and notification dissemination for an Internet-scale publish/subscribe system with a rich subscription model. We investigate the interface between event processing by a database server and notification delivery by a dissemination network. Previous research in publish/subscribe has largely been compartmentalized; database-centric and network-centric approaches each have their own limitations, and simply putting them together does not lead to an efficient solution. A closer examination of database/network interfaces yields a spectrum of new and interesting possibilities. In particular, we propose *message and subscription reformulation* as general techniques to support stateful subscriptions over existing content-driven networks, by converting them into equivalent but stateless forms. We show how reformulation can successfully be applied to various stateful subscriptions including range-aggregation, select-joins, and subscriptions with value-based notification conditions. These techniques often provide orders-of-magnitude improvement over simpler techniques adopted by current systems, and are shown to scale to millions of subscriptions. Further, the use of a standard off-the-shelf content-driven dissemination interface allows these techniques to be easily deployed, managed, and maintained in a large-scale system.

Based on our findings, we have built a high-performance publish/subscribe system named *ProSem* (to signify the inseparability of database processing and network dissemination). ProSem uses our novel techniques for group-processing many types of complex and expressive subscriptions, with a per-event optimization framework that chooses the best processing and dissemination strategy at runtime based on online statistics and system objectives.

Contents

List of Figures

List of Tables

Acknowledgements

My first and foremost wish is to thank my Ph.D. advisor, Jun Yang, for the valuable support and guidance that he has given me throughout my career as a Ph.D. candidate. He is responsible for helping me transform from a student unsure of how to perform research, to someone who is able to express ideas more clearly and do research in a more confident manner. I have learned a lot from Jun over the last several years: analyzing problems in a disciplined manner, writing clear and precise technical text, creating good presentations, and delivering them in a captivating manner. Whenever I have been stuck in research, either searching for interesting problems or groping for a direction toward solutions, Jun has always been there with his clear, precise, and incredibly useful advice. Thank you, Jun!

Special thanks go out to Shivnath Babu, Jeff Chase, Carla Ellis, and Hui Lei, who have served on my Ph.D. committee and have regularly provided helpful feedback on my work over the last several years. I had the fortune of interning for two summers under the mentorship of Hui Lei at IBM T. J. Watson Research Center, for which I extend my gratitude to him. I would also like to thank Pankaj Agarwal for his invaluable guidance and help in several matters. Special thanks also go out to Amin Vahdat, for his support during my initial years as a Ph.D. candidate. As part of the Computer Science Department at Duke University, I have had the chance to interact with some very talented and helpful people. In particular, I would like to mention David Becker, Chris Bond, Hao He, Richard Lucic, Jeff Phillips, Diane Riggs, and Adam Silberstein.

Some portions of this dissertation are based on my collaborations with other students. The server-side data structures for range-min subscriptions (Section 3.3) were developed in collaboration with Junyi Xie. An initial version of the B^A-tree(r) data structure for value-based notifications (Section 4.3.2) was designed in collaboration with Jeff Phillips. The coding of the client and visualizer interfaces of ProSem (see Chapter 6) was done in collaboration with Ying Zheng and Albert Yu.

I also deeply acknowledge the support of my family. I especially thank my parents for everything they have done for me and for inculcating the scientific temperament in me. I also thank God

for guiding and blessing me throughout this journey. Last but not least, I thank my wife Sita who stood strongly beside me through this journey, and made many sacrifices so that I may reach this goal. She is the wind beneath my wings, and I shall forever be indebted to her for that. Of course, no acknowledgement can be complete without mention of our beloved daughter Shruti, who gave me countless hours of joy, as well as a reason to graduate! Thank you!

Chapter 1

Introduction

With the advent of the Internet and Web 2.0, the necessity to capture and put together large amounts of information has brought about multiple large-scale data acquisition and integration efforts. An important next step is to disseminate data efficiently to users. Traditionally, users poll sources for information. However, polling too frequently may be inefficient, while polling less often may miss important updates. The data dissemination problem is exacerbated by the scale of the Internet, with potentially millions of interested users. The alternative push technology has long been the goal of publish/subscribe systems, which proactively push updates to users with matching interests. The push model is better suited for ensuring scalability and timely delivery of updates, important in many application domains: personal (e.g., RSS feeds, online auctions), financial (e.g., portfolio monitoring), security (e.g., reporting network anomalies, distributing critical software patches), etc.

Publish/subscribe is a model of data dissemination [FZ97], where data is aperiodically pushed from one or more sources (called *publishers*) to multiple destinations (called *subscribers*). Other dissemination models include periodic push (e.g., broadcast disks [AAFZ95]), periodic pull (client polling systems), and aperiodic pull (classic request/response client/server systems).

A publish/subscribe system (see Figure 1.1) decouples data providers or sources (publishers) from data recipients (subscribers). Publishers publish data to the system in the form of *events*, while subscribers specify their interests (i.e., what they wish to be notified of), in the form of queries called *subscriptions*. The underlying publish/subscribe middleware then assumes the task of matching data items to the relevant subscribers, and notifying them accordingly.

A typical publish/subscribe system is arranged as a set of networked machines or nodes called *brokers*, that cooperate to achieve the above goal. Subscribers attach themselves to brokers based on different criteria. Data sources generate updates to a database, which may be located at a server, a set of distributed servers, or distributed throughout the nodes in the network. The update could

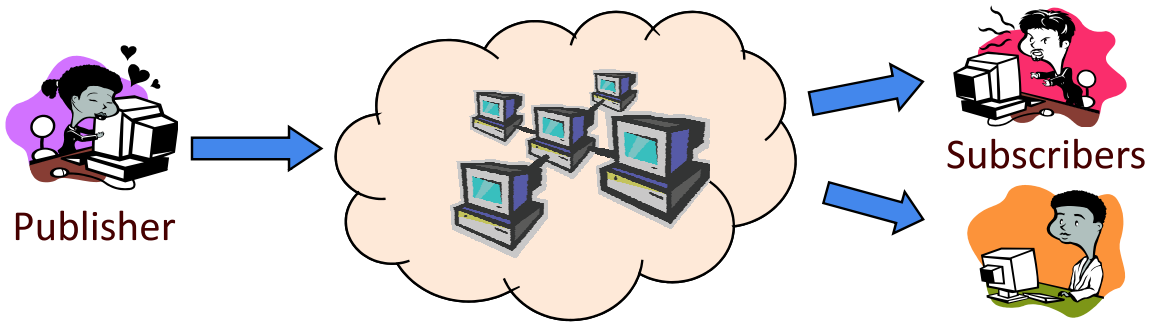


Figure 1.1: A publish/subscribe system.

affect a number of subscriptions. The publish/subscribe middleware is responsible for ensuring that the update is disseminated to all brokers hosting affected subscriptions.

1.1 Historical Views of Publish/Subscribe

The first generation of publish/subscribe systems, such as [OPSS93, Pow96, Sun], was based on *channels* (also referred to as subjects or topics). Channels are predefined (e.g., *sports*, *politics*, *entertainment*, etc.), and each update is tagged with the names one or more channels. Subscribers, on the other hand, specify a list of channels that they are interested in. The update is simply sent to all subscribers subscribing to that channel. These systems could take advantage of group delivery mechanisms such as multicast [CDKR02].

However, the granularity of channel-based publish/subscribe systems was often found to be too coarse to fit the particular interests of individual users. This inflexibility led to the second generation of *content-based* publish/subscribe systems. These systems support finer-granularity subscriptions defined as filters over the message contents. A large number of such systems have been built in recent years. We briefly sample several of these systems in Section 1.4. In a typical content-based publish/subscribe system, events conform to an *event schema* [OAA+00] that includes attributes such as `Symbol`, `Risk`, `Price`, `Earning`, etc. (in a stock trading application). Semantically, subscriptions are limited to being predicates over individual events, e.g., `Symbol` = "GOOG", or `Risk` $\in [20, 60]$ (between a particular range of risk ratings).

An incoming event can be matched to subscriptions either at a server or within the network. We call such subscriptions *stateless* because the matching can be performed without knowledge of any other state such as past history of events or the current subscription result. Content-based networks [CW01] are frequently used to disseminate events. Such systems have been extensively studied [ASS+99, PC05] in both the database and networking communities.

1.2 New Challenges in Publish/Subscribe

1.2.1 Towards Powerful Subscription Models

With an increasing number of data sources (e.g., RSS feeds, stock updates, network monitoring logs, etc.) as well as complex data demands in this Digital Age, data management has become a critical factor in large distributed applications. Users may want updates to be transformed, correlated, and/or aggregated. They may want to receive notifications only when certain important events occur, and want to receive data that are filtered, joined, and summarized. For example, consider a stock trading application:

Range-Aggregation Subscriptions Users may be interested in receiving information on stocks with minimum price-to-earning ratio (a popular measure of stock quality; lower is better), within specified ranges of risk factors. Such a subscription is useful because users are generally interested in high-performing stocks (having low price-to-earning ratio) within ranges of risk that they are comfortable with.

Subscriptions with Value-Based Notification Conditions Users may want stock updates, but only whenever the stock price changes by (say) 10% from the last notification. Such subscriptions are intuitively useful because users may want stock price updates, but do not wish to be inundated with every since change in price. They would like to precisely bound the granularity of receiving updates based on the current stock price, as compared to the price they last saw.

Select-Join Subscriptions Users may wish to correlate (join) information from multiple data sources. For example, they may want good analyst ratings (from one data source) for stocks with relatively high risk factor (from another data source). In general, when there are multiple data sources available to users, correlation between data sources becomes a natural user requirement.

The above categories of subscriptions are said to be *stateful*, because they may require us to look beyond the event itself in order to support the subscriptions. Specifically, we may need to look at other data items, subscription-specific state, or the event history. For example, in case of range-aggregation subscriptions in the stock trading application introduced earlier, we need to know what other stocks exist in the system in order to process a new event. In case of subscriptions with value-based notification conditions, we need to look at per-subscription state that includes the last value seen by the subscription. Select-join subscriptions also require us to look at other data items or the event history, in order to compute join results.

In a publish/subscribe system, the cost of supporting millions of subscriptions over a large network is extremely high, even for simple stateless subscriptions. More complex and expressive subscriptions in the form of database-style queries are much more challenging and are not well-supported by state-of-the-art systems. In both channel-based and content-based publish/subscribe systems, subscriptions are essentially simple stateless filters defined over individual messages, so they cannot express queries of interest across different messages, over the event history, or queries that require subscription-specific state to process. This limitation means that these systems cannot efficiently support modern application requirements such as those outlined earlier. They tend to generate excessive amounts of notifications, and require the end users to post-process notifications in order to filter out irrelevant ones. However, we desire that support for these powerful subscription models should not impact the system's scalability in terms of data rates, database size, and number of subscriptions.

1.2.2 Need for Designing the Right Database/Network Interface

At the conceptual level, the bulk of the work performed by a publish/subscribe system can be roughly divided into two components: 1) *subscription processing*, the task of matching and processing each incoming publish message with the large set of active subscriptions, and 2) *notification dissemination*, the task of notifying, over a network, those subscribers who are interested in the publish message. Traditionally, research in the database and networking communities has focused on subscription processing and notification dissemination in isolation.

At one extreme, database servers are directly responsible for notifying individual subscribers. Previous work from the database research community has focused on efficient subscription processing; notification dissemination is rarely addressed. Most existing work assumes that a server maintains the entire database state and all subscriptions in the system, and is responsible for computing the set of subscribers affected by each incoming publish message. A straightforward way to notify this set of subscribers is to unicast a notification to each of them in turn. When many subscribers need to be notified, this approach will incur a large amount of outbound traffic from the server, and may easily overwhelm the server and its network links. As server-side subscription processing techniques (such as sharing [CDTW00] and indexing [HCH$^+$99]) continue to mature, the dissemination bottleneck has surfaced in many systems, both research [DRF04] and commercial [KRE03].

At the other extreme, updates are injected directly into the network, and the network is solely responsible for processing subscriptions and forwarding notifications. The networking research community has always focused on efficient notification dissemination. Notable mechanisms include *multicast* [Agu84] and *content-based networking* [CW01]. With multicast, the system defines a number of *multicast groups*, each consisting of a set of subscribers, e.g., those who are interested in Google's stock. The network can efficiently disseminate the same message to all members of the group. With content-based networking, the system views each message as a tuple of attribute-value pairs. The network is typically implemented by an *overlay* of nodes that perform application-level routing. Each overlay node maintains a summary of all subscriptions reachable from each of its outgoing overlay links, and it forwards an incoming message onto an outgoing link if the message

matches the corresponding summary. Both mechanisms, however, support only stateless subscriptions that can be processed by examining the message itself. For multicast, a message's group id encodes its forwarding directions. For content-based networking, the message tuple contains all the information needed to forward the message.

Both the database-centric and network-centric extremes are unsuitable for large-scale expressive stateful subscription queries. Consider again the examples introduced in the stock trading application:

Range-Aggregation Subscriptions Assume that users are interested in receiving information on stocks with the minimum price-to-earning ratio (a popular metric of stock quality; lower is better), having risk factor in some user-specified range. On a stock update, a database-centric approach would *enumerate* the list of affected subscriptions at a server, and notify each using unicast. This can impose extreme stress and outgoing traffic at the server, introducing an unacceptable increase in the latency of notifying subscribers, and making it difficult to scale to many subscriptions. On the other hand, a simple network-centric approach might *relax* the subscriptions by dropping the MIN function. In other words, we could just inject the update into the network to reach every subscription whose risk factor range contains the updated stock without considering minimum. This creates unnecessary traffic if the update does not alter the minimum price-to-earning ratio for most subscriptions. In addition, this requires the end users to post-process notifications in order to update the original subscription result. In-network schemes to reduce traffic add complex application-specific logic into the network, making them difficult to manage and deploy, and they do not always work well.

Subscriptions with Value-Based Notification Conditions Consider an application that delivers stock price updates to a large number of subscriptions. While many subscribers may be interested in the same stock, each subscriber may have a unique data need in the form of a *notification condition*, which precisely specifies when the subscriber wishes to receive a price update. For instance, users may want stock updates, but only whenever the stock price changes by 10% from the last update. In the presence of millions of subscriptions, naive solutions do not scale: 1) Given

an incoming event, if we let a server check all subscriptions in turn and notify each affected one with unicast, processing and dissemination costs can easily overwhelm the server. 2) If we disseminate all events to subscribers and rely on post-processing at each subscriber to enforce notification conditions, there will be excessive network traffic, most of which is unnecessary.

Select-Join Subscriptions Users may wish to correlate (join) information from multiple data sources. For example, assume that there are two data sources. One source of data is basic stock information, with stock information including the price-to-earning ratio. Another source of data is analyst reports, including ratings and reviews by analysts. In general, a stock may receive multiple ratings from different analysts. A subscriber may be interested in cases where a stock's rating and its price-to-earning ratio respectively belong to two prescribed ranges—for example, good ratings for stocks with relatively high price-to-earning ratios. Such needs can be expressed as select-join queries. A simple scheme of computing joins and sending join results can incur excessive overhead due to the redundant information within and across messages. Purely network-centric approaches (e.g., [Agu84]) also carry high overhead and redundancy, and are unable to share processing and dissemination satisfactorily.

Thus, in order to enable smarter and scalable support for complex subscriptions in a wide-area publish/subscribe system, we need to look beyond the usual extremes of database-centric and network-centric processing, and search for new hybrid techniques that optimize for the dual goals of processing and dissemination. The unification of databases on one hand and publish/subscribe systems on the other hand, is the stepping stone towards supporting efficient Internet-scale data delivery for complex subscriptions.

1.2.3 Need for a Holistic Publish/Subscribe System Design

A crucial design requirement for a large-scale publish/subscribe system is that it should be easy to deploy and adopt by users. That is, it needs to be flexible enough, have a simple interface, and preferably use off-the-shelf networking components that make for easier deployment and mainte-

nance on a wide-area network.

One direction of recent research has been in the development of stateful publish/subscribe systems that work by augmenting the network of brokers with additional state and application-specific logic in order to support complex queries. Examples of such systems include *SMILE* [JS03, SDBL07] and *PADRES* [FJLM05]. In this dissertation, we focus on solutions that avoid this direction, because adding database state and associated application logic would increase system complexity, complicate failure and consistency issues, and create deployment hurdles as can no longer use an off-the-shelf networking component. Moreover, each new *subscription type* (group of subscriptions with the same basic query template) would require the development, deployment. and management of such state and application logic.

Many existing publish/subscribe solutions target specific classes of scenarios or subscriptions. However, there has not been much effort directed at integrating these techniques into a single framework. There are unique practical challenges in building a high-performance publish/subscribe system that supports multiple complex subscription types. It is not clear if we can develop a unifying framework to optimize support for such diverse subscription types, taking both subscription processing and notification dissemination into account in a holistic manner.

Moreover, oftentimes different strategies of processing and disseminating events may be optimal based on the subscription types and characteristics, event characteristics, system goals, and processing costs. For example, if very few subscriptions are affected by an event, it may be easier to send updates directly using a server instead of delivering them over a network of brokers. A high-performance system needs to adapt to the workload and system capabilities, while still providing scalable support for complex subscription types.

1.3 Contributions

In this dissertation, our goal is to address the scalability issues that arise when transforming Internet-scale publish/subscribe systems to support the next generation of expressive database-style subscription models and querying patterns. By performing smart data management with a novel holis-

tic approach incorporating both database processing and network dissemination, we can reduce costs and create powerful systems to support such subscriptions. Such systems are more efficient, use less network resources, and are ready to cope with the large scales and volumes of the data and queries of tomorrow. Specifically, we make the following contributions:

The Content-Driven Network Abstraction We propose an abstraction for wide-area data dissemination, called *content-driven networks (CN)*. Content-driven networks are a generic stateless class of dissemination networks in which data recipients are indexed as destinations. CN routes injected messages to their destinations, based on the message content alone. There are many well-studied dissemination schemes that fall under the umbrella of CN. CN gives a simple, stateless, off-the-shelf, easily deployable network substrate with a clean interface that hides the complexity of the dissemination layer from the publish/subscribe application. CN is described in detail in Chapter 2.

The Database/Network Interface Spectrum We investigate the interface between event processing by a database server and notification delivery by a dissemination network. Previous research in publish/subscribe has largely been compartmentalized; database-centric and network-centric approaches each have their own limitations, and simply putting them together does not always lead to an efficient solution. A closer examination of database/network interfaces yields a spectrum of new and interesting possibilities for the interface, each with their unique strengths and weaknesses. We describe the spectrum of interfaces in Chapter 2.

Message and Subscription Reformulation We propose *message and subscription reformulation* as general techniques to support stateful subscriptions over existing content-based networks, by converting them into equivalent but stateless forms. The basic idea behind reformulation is to compute semantic descriptions of affected subscriptions, and disseminate these descriptions (with the associated content) using CN. Designing the right description is non-trivial and often requires new ways of looking at the problem along with novel re-writing of subscription queries. Moreover, computing the descriptions efficiently requires novel data structures and algorithms. We discuss the

idea of reformulation in detail in Chapter 2, to lay the foundation for the rest of this dissertation.

Applying Reformulation to Various Subscription Types We apply reformulation to efficiently support various types of stateful subscriptions including range-aggregation (min, max, sum, count, average, distinct) in Chapter 3, subscriptions with value-based notification conditions in Chapter 4, and multi-way select-joins in Chapter 5. In each case, we first suggest several alternate solutions at different points on the spectrum of database/network interfaces. Armed with our idea of reformulation, we provide new insights into each problem, design novel schemes to solve the problems, and propose new algorithms and I/O-efficient data structures to implement the schemes. Extensive experiments demonstrate that these techniques can provide orders-of-magnitude improvement in both network costs and server processing time, with scalability to millions of subscriptions. We also demonstrate how these ideas can also improve the scalability of traditional continuous query processing in many cases, by helping us identify the affected queries very efficiently. An important finding is that one does not have to pay for improved network performance with degraded server costs — with careful design, it is possible to obtain orders-of-magnitude improvement in both metrics.

Improving the Performance of CN We propose a novel extension to CN called CN*, which addresses a source of inefficiency unique to stateful subscriptions, where the same bulky data may need to be disseminated to different subscriptions at different points in time due to the stateful nature of the subscriptions. CN* is application-agnostic and incrementally deployable, and reduces the dissemination redundancies inherent in many complex stateful queries by up to an order of magnitude. We describe CN* in Chapter 5, in the context of select-join subscriptions.

The ProSem Publish/Subscribe System Based on our research findings, we design and develop a scalable high-performance next-generation publish/subscribe system called *ProSem* (to signify the inseparability of processing and dissemination). ProSem uses a novel holistic approach that employs 1) CN as the stateless, easily deployable dissemination interface, and 2) specialized database processing to support many types of complex stateful subscriptions using this clean state-

less network interface. The simple dissemination interface enables easy deployment on a large scale, while the processing techniques using customized algorithms and I/O-efficient data structures reduce server costs and allow scaling to millions of subscriptions at low network cost. ProSem addresses the unique optimization challenges that arise in the context of supporting a wide range of subscription types and several processing/dissemination schemes, and proposes a new per-event optimization framework that chooses the best *plan* at runtime based on online statistics and user-controllable system objectives. ProSem effectively treats different server processing and network dissemination techniques as indexes (with the associated costs) whose usage is driven by the optimization framework. ProSem is discussed in detail in Chapter 6.

1.4 Related Work

In this section, we cover related work that is broadly related to this dissertation. The related work that are specific to each distinct contribution of the dissertation are covered in their respective chapters.

Background A survey of data-dissemination systems can be found in [FZ97]. We adopt the publish/subscribe model of dissemination, which is based on aperiodic push. Other models include periodic push (e.g., broadcast disks [AAFZ95]), periodic pull (client polling systems), and aperiodic pull (classic request/response client/server systems).

Original Publish/Subscribe Systems The first generation of publish/subscribe systems were channel-based, for example, [OPSS93, Pow96]; the publish/subscribe feature of Java Message Service (JMS) [Sun] also falls into this category. Channels are predefined and their granularity is often too coarse to fit the particular interests of individual users.

A large number of such second-generation content-based publish/subscribe systems have been built in recent years. *SIFT* [YGM99] is a publish/subscribe system for text documents; subscriptions are keyword-based filters on newly published documents. XFilter [AF00] and Web-Filter [PFJ+01] are publish/subscribe systems for XML documents, where subscriptions are de-

fined as XPath filters on individual XML documents. The *SIENA* wide-area event notification service [CRW01] supports a standard form of subscriptions used by most work on content-based publish/subscribe; these subscriptions are arbitrary boolean predicates over contents of individual messages, which are record- or object-structured. Hermes [PB02] uses content-based message delivery over peer-to-peer overlay networks. *REBECA* [MÖ1] is another publish/subscribe systems that supports content-based routing. SemCast [PC05] is a recent system that works with either XML or relational data.

In all these systems, subscriptions are mostly restricted to stateless filters or queries defined over individual messages, so they cannot express queries of interest across different messages or over the event history. Filters are not powerful enough to accommodate stateful database-style subscription requirements.

Stateful Publish/Subscribe Systems A number of systems built by the database community have made the subscription language more powerful. XML Message Broker [DF03] and *ONYX* [DRF04] additionally support on-the-fly transformation of XML according to a subset of XQuery; on the other hand, filtering and transformation are still limited to individual messages. Xyleme [NACP01], another publish/subscribe system for XML, provides a more powerful subscription language, supporting both monitoring queries and continuous queries (analogous to our notification conditions and subscription queries). However, their monitoring queries can refer only to individual updates but not to the database state. This limitation makes it impossible to express, for example, a notification condition based on a user-defined error metric. Cayuga [DGH$^+$06] is a publish/subscribe system that supports stateful subscriptions in the form of complex events. Cayuga scales well with event rates and number of queries, but does not address the problem of disseminating results to a large number of users located over a wide-area network. A few continuous query systems, which we will discuss in more detail below, also support rich query languages. Unfortunately, while the above systems support more powerful subscriptions, they tend to be database-centric and do not address the problem of efficiently delivering updates to subscribers over a network.

ONYX [DRF04] has begun addressing this problem; however, the focus of ONYX on support-

ing transformation of XML messages is quite different from our goal of supporting stateful subscriptions that cannot be processed on individual messages. *SMILE* [JS03] supports SQL queries and considers delivery, but pushes application-specific logic into the network, and makes the system design complicated and difficult to deploy and maintain on a large scale. It also uses a weak eventual consistency model that may not be suitable for many applications. *PADRES* [FJLM05] is a publish/subscribe system which supports composite subscriptions that can detect event patterns. While stateful, PADRES also introduces application-specific logic and transformations inside the network, which complicates deployment and management.

Complex Subscription Processing Continuous query systems [TGNO92, LPT99, CDTW00] and stream processing systems [DEB03] can be regarded as a form of publish/subscribe in which continuous queries over streams correspond to subscriptions. By default, these systems notify whenever the subscription content changes. OpenCQ [LPT99] supports notification conditions that refer to current and previous database states, and NiagaraCQ [CDTW00] supports timer-based conditions. However, they do not optimize subscription processing and notification dissemination jointly.

The idea of group processing has been identified and used in trigger processing and continuous query processing systems [HCH+99, CDTW00, YSG03]. Work on scalable database trigger processing [HCH+99] focuses on exploiting common patterns in triggering conditions. Work on scalable continuous query processing [CDTW00, MSHR02, CF02] focuses on exploiting common patterns in continuous queries. In particular, predicate and query indexing techniques have been developed in [HCH+99, CDTW00, FJL+01, YSG03, LLL+06] to speed up group processing.

Subscriptions can also be seen as materialized views, a well-known area of research in database systems [GM99a]. Instead of recomputing subscription contents from scratch every time, we can evaluate them incrementally. Thus, techniques for incremental view maintenance [GMS93, RCK+95, GL95, Qua96, CGL+96, CKL+97, YW98, LYC+00, PSCP02] can be adapted for subscription processing, though they do not apply directly to group processing of a large number of subscriptions.

Notification Dissemination The problem of efficient message delivery has long been tackled in networking and distributed systems research. The traditional delivery mechanism is based on client polling, which is susceptible to missing important updates. The next generation of delivery mechanisms uses real push techniques based on IP unicast or group-based multicast protocols, e.g., IP multicast. IP unicast, while widely supported, does not scale with the number of subscriptions. Multicast provides a perfect interface for channel-based subscription services. IP multicast has also been exploited in building publish/subscribe systems that support more general filter-style subscriptions [OAA+00]. Because of the slow adoption of IP multicast, there have been many recent proposals for supporting multicast at the application level by using an overlay network (e.g., [RHKS01, ZZJ+01, CDKR02]). Oftentimes, these overlay networks implement a distributed hash table (DHT) interface, which provides a convenient abstraction for addressing data in the network (e.g., [RD01a, RFH+01, SMK+01, ZKJ01]).

An alternative dissemination interface is content-based networking [CW01, ASS+99, CF03a]. Content-based networks have been well studied and can be used directly to implement a publish/subscribe system supporting filter subscriptions. Many such systems have been developed based on DHTs (e.g., [TA04, TAJ03, TBF+03, GSAA04]). However, their subscriptions are limited to simple stateless filters on update streams.

Chapter 2

Database/Network Interfaces and Reformulation

The goal of this dissertation is to enable support for complex and stateful distributed subscriptions in large-scale publish/subscribe systems. Recall that a publish/subscribe system has two basic components: subscription processing at a database, and notification dissemination over a network. In Chapter 1, we motivated the need to examine the spectrum of possible interfaces between the database and the network. In this chapter, we list the set of possible points in this spectrum and examine each point in detail. We use range-min subscriptions as the running example to illustrate each of the points in the spectrum. In the context of this spectrum, we also cover the concept of *content-driven networks* and our proposed technique of *reformulation*.

2.1 Motivation

In traditional publish/subscribe systems, subscriptions are essentially simple stateless filters defined over individual messages, so they cannot express queries of interest across different messages or over the event history. This limitation means that these systems cannot efficiently support modern application requirements. They make the end users deal with complex local post-processing, which requires maintaining unnecessarily large amounts of state and results in high volumes of updates.

As motivated in Chapter 1, modern applications and users may want updates to be further transformed, correlated, and/or aggregated. They may want to receive notifications only when certain important events occur, and want to receive data that are filtered, joined, and summarized. For example, with a range-min subscription, a user can track the minimum `PER` (price-to-earning ratio, a popular measure of stock quality) of stocks within a risk range. This subscription is stateful, because just by looking at a stock update message, the system cannot always tell whether or how the message would affect the subscription. To meet the needs of such applications, we need a pub-

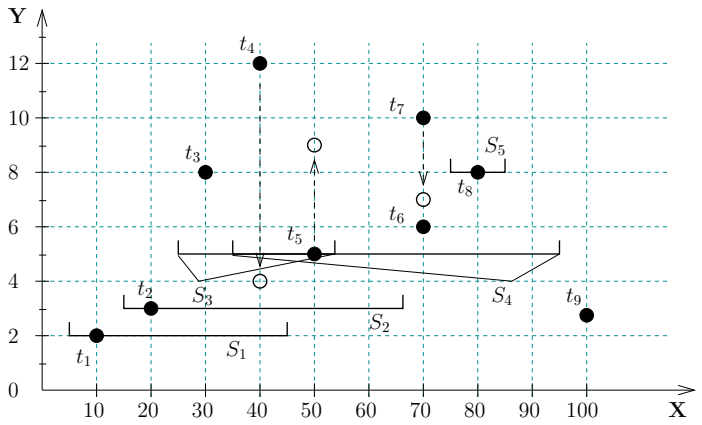


Figure 2.1: Example `Stock` table and range-min subscriptions.

lish/subscribe interface that supports a powerful subscription model that incorporates rich content definitions and flexible notification conditions.

To better illustrate the challenges in supporting richer subscription models using traditional techniques, consider again the example of range-min subscriptions (we use this class of subscriptions as a running example throughout this chapter).

Example 1 (Range-min subscriptions)**.** *Consider a publish/subscribe system that monitors the stock market for a large number of traders over a wide-area network. Conceptually, the system provides a database view* `Stock`$(\texttt{Symbol}, \texttt{Risk}, \texttt{PER}, \ldots)$ *that continuously tracks the up-to-date information for each stock. Suppose that a user is interested in tracking the minimum* `PER` *of stocks within a risk range* $[x_1, x_2]$ *that she is comfortable with. She can define a subscription over* `Stock` *using a SQL query:* `SELECT MIN(PER) FROM Stock WHERE` x_1 `<= Risk AND Risk <=` x_2*. To simplify discussion, let us focus on updates of* `PER`*, and assume that each update message has the schema* $\langle \texttt{Symbol}, \texttt{Risk}, \texttt{PER}, \ldots \rangle$*, where* `PER` *is the new price-to-earning ratio after the update. When such a message arrives, the system needs to notify those users whose subscription query results are affected by the* `PER` *update.*

The range-min subscription is stateful. To illustrate, consider the current state of `Stock` *shown as a collection of points (labeled by* t_i *and shown in solid black) in Figure 2.1; the* X*-axis plots* `Risk`*, while the* Y*-axis plots* `PER`*. Each range-min subscription (labeled by* s_i*) is represented as a horizontal interval spanning the risk range of interest, whose height equals the minimum* `PER` *in*

that range.

Suppose that an update lowers t_4's `PER` *to just below that of t_5 (indicated by a dotted line with arrow). This update should affect subscriptions s_3 and s_4, but not s_1, s_2, or s_5. For s_1 through s_4, their ranges all cover t_4's* `Risk`*. In order to determine that s_3 and s_4 are affected while s_1 and s_2 are not, the system must be able to compare t_4's new* `PER` *with the minimum* `PER` *currently maintained by each of these subscriptions; this latter information is not available in the update message. A more complicated situation arises when the current minimum* `PER` *shared by a group of subscriptions is updated higher, potentially exposing them to different new minima. For example, suppose that an update raises t_5's* `PER`*, as illustrated in Figure 2.1. As a result of this update, s_3 should be updated with t_3's* `PER`*, while s_4 should be updated with t_6's* `PER`*. Neither piece of information is available in t_5's update message. In general, the system must maintain the entire state of the* `Stock` *table in order to handle such updates.*

Example 2 (Supporting range-min subscriptions)**.** *Following the traditional database-centric approach, we can use a server to maintain all subscriptions and the up-to-date state of the* `Stock` *table. Thus, for each incoming stock update message, this server can easily compute which subscriptions are affected and how they need to be updated. However, options for disseminating these notifications are limited. (1) Unicast is the most natural way, but can be inefficient for a large number of affected subscriptions. (2) Content-based networking is difficult to leverage because of an "impedance mismatch": A content-based network performs matching between messages and subscriptions in the network, while in this database-centric approach the server has already computed the list of affected subscriptions; converting this list back to a message for dissemination in a content-based network is not straightforward, and would be a waste of resources because of duplicate processing. (3) Multicast is a possibility, but to do a perfect job, we would need a multicast group for every possible subset of the subscriptions that could be affected by a stock update in the same way. There may be a prohibitively large number of such groups (up to 2^m if every subset from a total of m subscriptions can form a group), rendering multicast infeasible. Even if we restrict the problem to range-min subscriptions alone, it is unclear how to reduce the space of possible groups. For example, in Figure 2.1, although s_2's risk range contains that of s_3, not every update affecting*

s_3 would affect s_2, and vice versa.

An alternative is to follow a network-centric approach. Content-based networking is a natural starting point because it supports range subscriptions. We can "relax" a range-min subscription `SELECT MIN(PER) FROM Stock WHERE` x_1 `<= Risk AND Risk <=` x_2 *to a range subscription* `SELECT * FROM Stock WHERE` x_1 `<= Risk AND Risk <=` x_2*. The network would then forward to each subscriber every stock update message that falls within her risk range. Each subscriber locally maintains the content of the range subscription from which the range-min subscription can be derived. Note that all stocks in the range must be maintained (not just ones with the minimum* `PER`*) in order to handle the case when the minimum* `PER` *rises. Besides this maintenance overhead, a more serious issue is that the relaxation of a stateful subscription into a stateless one can potentially result in much more update traffic. For example, in Figure 2.1, any* `PER` *movement of t_4 above t_5's* `PER` *has no effect on any subscriptions, but with this approach, all updates of t_4 would still be forwarded to s_1 through s_4 simply because t_4 falls into their risk ranges.*

The above examples show that efficient support of stateful subscriptions is a challenging problem for wide-area publish/subscribe systems. On one hand, existing network dissemination mechanisms do not support stateful subscriptions directly. While it is possible to relax a stateful subscription into a stateless one and rely on subscribers to perform additional local post-processing, doing so requires unnecessarily large amounts of local subscription state and high volumes of notifications. On the other hand, while the database-centric approach can easily process stateful subscriptions at a server, disseminating notifications over a wide-area network remains difficult because of the inefficiency of unicasts and the difficulty in interfacing the server with advanced network dissemination mechanisms such as multicast and content-based networking.

We argue that the key to the solution lies in properly *interfacing* the database with the network, in order to combine the processing power of database servers and the dissemination power of the network effectively. In general, there is a wide spectrum of possibilities for interfacing the database with the network and for dividing up work between them. These possibilities provide an interesting set of trade-offs in terms of efficiency, scalability, and manageability of the system. To the best of our knowledge, there is no prior work that investigates this spectrum of database/network

interaction models comprehensively. This unified perspective from both databases and networking enables us to identify interesting hybrid solutions that outperform approaches that are either database-centric or network-centric.

To recap, the combination of (1) cooperative processing and dissemination by the database and the network, (2) a clean, easy-to-implement database/network interface, and (3) efficient server-side data structures and algorithms together provide an efficient platform for supporting stateful subscriptions over a wide-area network.

The remainder of this chapter is organized as follows. Section 2.2 introduces our new model for publish/subscribe systems supporting complex subscriptions, and discusses the concept of content-driven networks in detail with several examples. Section 2.3 discusses various methods for interfacing the database with the network, including database-centric, network-centric, and hybrid approaches. Our novel reformulation technique is introduced in the context of the hybrid approach (Section 2.3.3). We then cover the details of reformulation in general, in Section 2.4. There exists a spectrum of approaches to reformulation itself, and these are covered in Section 2.5.

2.2 Content-Driven Networks

2.2.1 Our Model of Publish/Subscribe

A publish/subscribe system is responsible for delivering data generated by publishers, to interested subscribers whose interests over data are defined as subscriptions. In large-scale, wide-area publish/subscribe system, the number of subscriptions is typically high, and subscribers can be located all over the network. Traditional publish/subscribe systems assume that events follow some schema, and subscriptions are filters over individual events (e.g., `PER` $\in [45, 70]$ for a `Stocks` event)

We use a publish/subscribe model that supports more powerful database-style subscriptions. We assume that a central server maintain a database. Publishers of data send events to the server. Events are modifications (inserts, deletes, and updates) to the database. Let $\mathcal{D}$ indicate the current database state at the server. A subscription is a query Q over the database. On creation of the subscription,

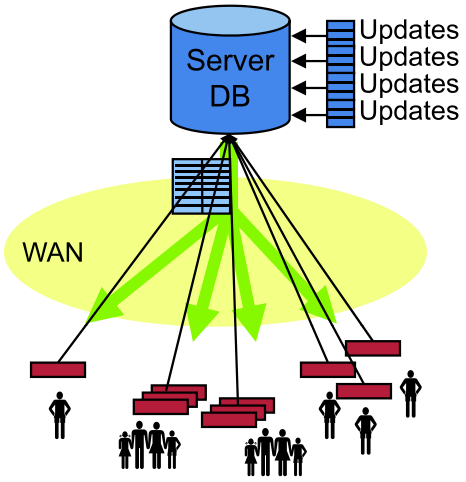


Figure 2.2: Our publish/subscribe model.

the server delivers an initialization message to Q's subscriber, such that $Q(\mathcal{D})$ can be computed from the content of this initialization message. On an incoming event, the database state changes from $\mathcal{D}$ to $\mathcal{D}'$. If $Q(\mathcal{D}') \neq Q(\mathcal{D})$, and we say that the subscription Q is *affected* by the event; the task of the publish/subscribe system is to deliver a notification message to Q's subscriber, such that $Q(\mathcal{D}')$ can be computed from $Q(\mathcal{D})$ and the content of this notification message. This model is similar to the *view maintenance* [GM99a, AASY97, ZGMHW95] setting, but with the problem of scaling to a large number of diverse views distributed over a wide-area network. Figure 2.2 depicts our model of publish/subscribe.

The simplest scheme of notification delivery under this model is to send messages directly from the server to each affected subscriber, using mechanisms such as IP unicast. This scheme precludes the sharing of delivery across subscriptions. In order to share the costs of notification delivery across subscriptions, we use a network of *brokers*, like most wide-area publish/subscribe systems (as discussed in Section 2.1). Each broker is assigned a subset of the subscriptions. The brokers collectively forward notification messages among themselves and to the subscribers. The last hop of notification delivery, from a broker to an affected subscription assigned to it, can use any delivery mechanism suitable to the subscription client, e.g., IP unicast, emails, or instant messages.

2.2.2 Overview of CN

We propose a new abstraction as a technique for building the broker network, called a *content-driven network* (*CN*).[1] CN refers to a class of overlay networks for data dissemination. CN has a clean and simple dissemination interface. Each message contains a list of attribute-value pairs. A subscription is a filter over individual messages, defined as a predicate involving message attributes. CN efficiently routes each message to all matching subscriptions.

The duality of query and data allows us to view CN from a different angle. In the view above, destinations interests (subscriptions) are queries and messages are data. Alternatively, we can regard subscriptions as data distributed in the network, and messages as queries that need to reach the relevant data.

In this work, we treat CN as a black-box delivery mechanism. Many structured as well as unstructured overlay networks can be classified as CN. We next describe several instances of CN with varying degrees of expressiveness in the predicates they support. Users can choose an appropriate CN, depending on convenience and the desired level of expressiveness (application-dependent).

2.2.3 Content-Based Networks

Content-based networks (*CBN*) [CW01] are perhaps the most general incarnation of CN. They support messages with arbitrary attributes and destinations with interests expressed as arbitrary boolean predicates involving message attributes. There are many well-established and popular systems that use content-based networking as the dissemination layer, e.g., SIENA [CRW01], Gryphon [OAA+00], REBECA [MÖ1], Hermes [PB02], PADRES [FJLM05], JEDI [CNF01], XNet [CF03a], etc. Thus, content-based networks are easy to configure, deploy and maintain in a large-scale system.

[1] Terms such as *content-based routing*, *content-based networking*, and *semantic multicast* capture similar concepts. We choose not to use these terms because they are often associated with specific projects and systems, e.g., [CW01, CRW01, PC05]; we want to capture a broader class of systems with different designs and varying degrees of expressiveness.

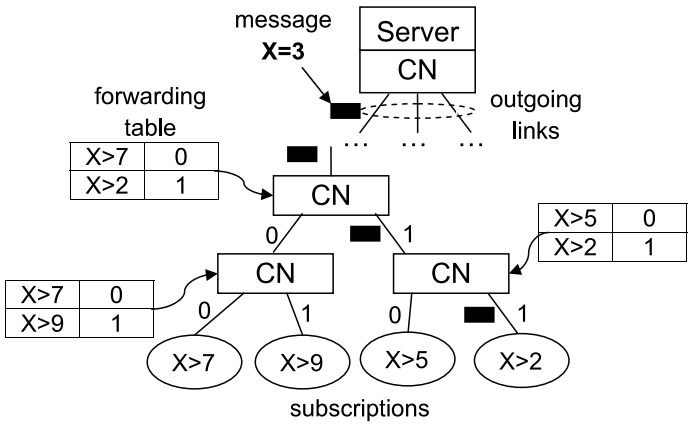


Figure 2.3: Using a content-based network for publish/subscribe.

A content-based network can accept messages to be forwarded to a set of matching *destinations*. A message is a set of typed attributes following some *schema*. For example, a message M following a schema that includes `Symbol` and `Price` as two of its attributes, might look like $\langle$`Symbol`: "GOOG", `Price`: $550, \ldots\rangle$. Destinations, on the other hand, are simple predicates over the message attributes. For the example schema above, a destination may be a set of predicates such as (`Symbol` = "GOOG") $\wedge$ (`Price` $\in [500, 600]$). Notice that the message M matches this destination, and therefore M would be delivered to this destination.

Operational Details: The network is responsible for delivering to each destination, every message matching the predicates declared by that destination. Message delivery is performed in a multi-hop manner over an overlay network of nodes. The delivery function consists of two interrelated subfunctions: routing and forwarding. Routing amounts to establishing flow paths through the network by compiling and positioning local forwarding tables at each node. A forwarding table contains the information necessary for a node to decide to which neighbor node or nodes a given message should be sent; the processing of a message at a node is the forwarding subfunction. Taken together, the forwarding performed at the nodes causes messages to be routed through the network, until they reach all the affected destinations. Content-based networks use sophisticated techniques [CRW01, OAA$^+$00] to build concise forwarding tables at the nodes, and to perform the forwarding function efficiently. Figure 2.3 shows the routing of a message using forwarding tables,

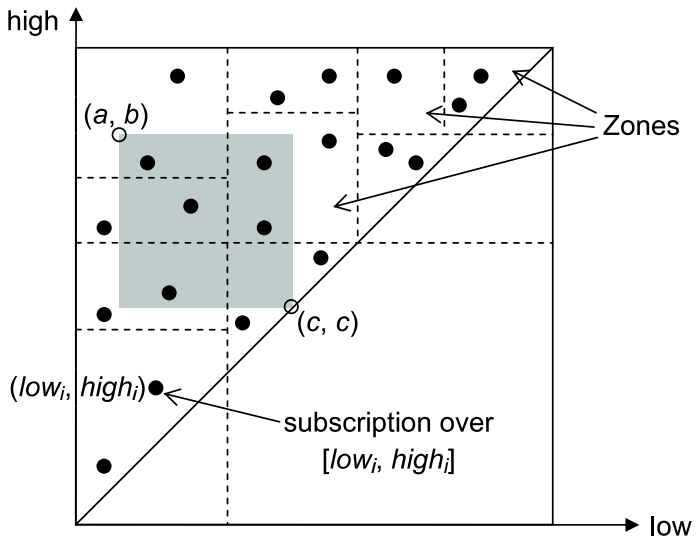


Figure 2.4: Using a CAN for publish/subscribe.

for 4 subscriptions with predicates on an attribute X, installed in the system.

2.2.4 Content-Addressable Networks

Another example of CN, which supports less expressive subscriptions, is a *content-addressable network* (*CAN*) [RFH$^+$01], where a message contains d numeric attributes and each destination node implicitly specifies an orthogonal range predicate in the d-dimensional space as its subscription. A CAN is a decentralized structured overlay network that uses a logical d-dimensional Cartesian coordinate space that is partitioned across all participating peers. Each peer is responsible for a subspace in the form of a hypercube, called a *zone*. Each zone has knowledge of only its neighbors and can route messages only to them. Routing from a source point to a destination point in the CAN space is carried out in multiple hops until the destination is reached.

We can use a CAN to build a structured stateless dissemination layer for a publish/subscribe system. Assume that each subscription is mapped as a point in the CAN space. For example, if subscriptions can specify range selection predicates over k attributes, we index each subscription as a point in a $2k$-dimensional CAN as follows. Each attribute used in range selection is mapped to two dimensions in the CAN space, one for the low end of the range and the other for the high

end. A range subscription can then be represented as a point in this space. Figure 2.4 illustrates a 2-d CAN, where each single-attribute range subscription over the range $[low_i, high_i]$ is mapped to the point $(low_i, high_i)$ in the CAN space. Note that in general, a subscription could be mapped to a point in d-dimensional CAN space based on the values of any d subscription-specific parameters (not necessarily range predicates).

The CAN space is partitioned into rectangular *zones*, each with a *zone owner*—a network node responsible for all the subscriptions in its zone. Partitioning of the CAN space into zones can use load balancing criteria, for example, the number of subscriptions residing in the zone and the number of events reaching the zone. The zone owners correspond to brokers in the publish/subscribe system. In order to use the CAN-based CN for reaching affected subscriptions, we inject a message with an attached description into CN. The description can be interpreted as an arbitrarily complex *region* in CAN space that we wish to reach. Every subscription lying within the region is considered affected by the message. CAN routing can easily be adapted to reach all zones within a specified region, by first routing to some corner of the region and then proceeding to forward messages from zone to zone until we cover the entire region.

Expressiveness

If subscriptions are mapped to the CAN space based on their range predicates, a hypercube region in d-dimensional CAN space can succinctly express a conjunction of d predicates , where each predicate specifies either (1) containment of a subscription's range predicate within a specified range, or (2) containment of a specified range within a range attribute of a subscription. More complex region definitions can express arbitrary containment predicates succinctly.

For example, a CN message $\langle \ldots$,UL_X: a, UL_Y: b, LR_X: c, LR_Y: $c\rangle$ may be interpreted as a rectangular region (in 2-d CAN) with upper-left coordinate (a, b) and lower-right coordinate (c, c), shown shaded in Figure 2.4. This description identifies every subscription whose range predicate 1) is contained within the range $[a, b]$, and 2) contains the range $[c, c]$. Equivalently, in a CBN we could rewrite every subscription X_i with range predicate $[low_i, high_i]$ as the predicate $([low_i, high_i] \subseteq [\texttt{UL\_X}, \texttt{UL\_Y}]) \wedge ([\texttt{LR\_X}, \texttt{LR\_Y}] \subseteq [low_i, high_i])$. Note that unlike a CBN, a CAN-based CN cannot

support arbitrary predicates, including keyword matches and user-defined functions.

An Example: Meghdoot

Meghdoot [GSAA04] is a publish/subscribe system that uses a simpler version of the CAN-based dissemination layer just described. Meghdoot only supports subscriptions with range predicates, with simple stateless matching of events to subscriptions matching all range predicates. Meghdoot uses CAN space to notify affected range subscriptions as follows. Consider an update to a tuple whose range attribute value is c. The affected range subscriptions are precisely those in the upper-left quadrant rooted at the point (c, c) (on the diagonal). Subscriptions inside this quadrant are affected because their ranges contain c; subscriptions outside this quadrant are unaffected because their ranges do not contain c. Hence, Meghdoot first routes the update message towards the point (c, c) which is called the *event point*; the zone owner of that point then forwards this message to the neighboring zone owners in the affected quadrant, which in turn forward the message upward and leftward to their neighbors, and so on. CAN provides a convenient substrate to implement this forwarding algorithm. Further details of this algorithm can be found in [GSAA04].

2.2.5 Other CN Instances

Many dissemination mechanisms and overlay networks can be considered to fall under the CN umbrella. These include multicast networks (e.g., [CDKR02, PC05]) and distributed indexes such as prefix hash trees [CRR$^+$05], P-trees [CLGS04], BATON [JOV05], SD-Rtrees [dMLR07], etc. However, these substrates have considerably lower expressiveness. For example, a multicast network supporting multiple multicast groups can be seen a CN because messages carry a group id attribute. Destination interests, implied by group memberships, can be regarded as message predicates that select particular group ids. Distributed one-dimensional range search indexes (e.g., prefix hash trees [CRR$^+$05]) are CN, because we can regard a node responsible for data item s as interested in all range search messages satisfying the predicate $(\texttt{S\_L} \leq s) \wedge (s \leq \texttt{S\_R})$, where S_L and S_R are the two message attributes corresponding to the left and right endpoints of the search range.

2.2.6 Discussion

While the simplicity of CN's network interface enables efficient and scalable implementations, subscriptions directly supported by CN are limited to predicates (usually selections) over individual event tuples. CN can directly support notification dissemination for stateless subscriptions, as long as the predicates defining the subscriptions are supported by the particular network used. We can simply inject each event into the CN, and it will deliver the event to all destinations interested in this event in an efficient manner. Note that in case of CN instances based on CAN or distributed indexes, overlay nodes cannot specify their own interests; their predicates are chosen by the network. In these cases, to support subscriptions with arbitrary interests, we need to map subscriptions to appropriate overlay nodes, which serve as brokers responsible for forwarding events to them.

Compared with unicast-based notification dissemination, CN offers better scalability, and incurs much less network traffic when lots of subscribers need to be notified. However, CN cannot be directly used for precisely supporting stateful subscriptions such as range-aggregation and select-joins. The reason is that for stateful subscriptions, processing requires information beyond individual messages—we cannot determine, just by examining the content of an incoming event message and a subscription definition, whether the subscription is affected by the event (i.e., whether the subscription needs to be notified because of the event).

2.3 Spectrum of Database/Network Interfaces

This section explores the spectrum of possibilities for interfacing servers with a network in order to support stateful subscriptions efficiently. We start with a brief discussion of the database-centric approach in Section 2.3.1. Then, Section 2.3.2 discusses the network-centric approach. Section 2.3.3 describes an important contribution of this chapter—a hybrid approach that supports closer cooperation between servers and the network using message/subscription reformulation.

To keep our discussion focused, we use range-min subscriptions as a running example. To make our discussion concrete, recall from Example 1 the database table

$$\texttt{Stock}\,(\texttt{Symbol}, \texttt{Risk}, \texttt{PER}, \ldots)$$

and range-min subscriptions of the form

$$\texttt{SELECT MIN(PER) FROM Stock WHERE } x_1 \texttt{ <= Risk AND Risk <= } x_2.$$

We call Risk the *range attribute* and PER the *aggregation attribute*. To simplify discussion in this chapter, we further restrict ourselves to decreasing updates of the aggregation attribute (PER) only. We consider range-aggregation in general, in Chapter 3.

Range-aggregation subscriptions are useful in many situations where users are interesting in tracking the "best" objects in ranges of their interest, e.g., stocks with the lowest price-to-earning ratios within a risk range, or lowest-priced digital cameras with at least 4.0 megapixels. The various database/network interface approaches and the message/subscription reformulation mechanism that we are going to present later are completely general; however, the actual reformulation technique may vary for different subscription types. We discuss the details of how to handle various subscription types in subsequent chapters.

2.3.1 Database-Centric Approaches

In this set of approaches, we follow the traditional database-centric view of publish/subscribe—of first computing the updates to each subscription, and then disseminating these updates. We assume that a single server maintains the database state and keeps track of all subscriptions. For each publish message, we can efficiently compute all subscription updates using group processing techniques such as those developed for continuous query processing [LPT99, CDTW00, TGNO92]. In case of range-aggregation, we have developed group-processing techniques (to be presented in Section 3.3) that can compute all subscription updates in time sublinear in the size of the database and the number of subscriptions. The approaches below differ mainly in how subscription updates are disseminated.

S-UN: Server with Unicast Network

With this approach, which we call S-UN, the server unicasts a subscription update message to each affected subscription. For our running example, the message has a constant size, and simply contains the new minimum PER for the subscription. The problem with this approach is that when many subscriptions are affected, there will be a large amount of traffic overall, and the server can easily become a bottleneck of dissemination.

If multiple affected subscriptions are hosted by the same node in the network, an additional optimization is for the server to combine multiple messages into one. This technique, which we call *message aggregation*, reduces the number of messages. However, the size of a combined massage would no longer be constant; instead, it becomes linear in the number of affected subscriptions at the node. The reason is that the message needs to list the affected subscriptions (because not all subscriptions at the node may be affected) and possibly multiple subscription updates (because different subscriptions may be affected differently by the same database update, as shown in Example 1).

S-MN: Server with Multicast Network

Multicast [Agu84] is an efficient mechanism for disseminating messages to a group of network destinations. Ideally, we would define a multicast group for each subset of the subscriptions. After the server computes all subscription updates, it checks to see which subset of the affected subscriptions share the same update message, and sends out this message to the multicast group consisting precisely of these subscriptions. However, both IP multicast [Agu84] and application-level multicast [CDKR02] techniques do not handle the need for large number of groups (up to 2^M for M subscriptions).

In this work, we resort to *hierarchical application-level multicast*. This method builds a tree rooted at the server spanning all nodes hosting subscriptions, with a moderate fan-out c. Each non-leaf node and its children together form a multicast network with 2^c multicast groups, each supporting efficient application-level multicast from the node to a subset of its children. Thus, this method avoids the problem of having too many multicast groups by breaking down the dissemina-

tion task into a hierarchy of much smaller multicasts with group number capped at 2^c. The cost of doing so is that the update message sent out by the server must list the set of affected subscriptions; otherwise, a non-leaf node would not be able to tell which children to forward the message to.

We call the above approach S-MN. The problems with using multicast for publish/subscribe (large number of groups and large message size) have also been identified by other work [BCM+99, OAA+00, PC05]. Possible solutions are: (1) Reduce number of groups [OAA+00] by approximating group membership in which case post-processing and additional state are needed at subscribers. (2) Use compact, "semantic" descriptions of affected subscriptions [BCM+99, PC05] to avoid large update messages. This approach gives a content-driven network (that still does not handle stateful subscriptions), which we consider next.

2.3.2 Network-Centric Approaches

At the other end of the spectrum, we have approaches that avoid the use of servers altogether by making the network handle as much subscription processing as possible. A natural starting point is a content-driven network (CN), which supports stateless subscriptions defined as predicates over the content of each message. Information about subscriptions is reflected in the distributed routing state of the network, which allows an update to be forwarded to affected subscriptions without intervention of servers. We use the CAN-based CN (Section 2.2.4) as a running example in this chapter. We next explore how to support stateful subscriptions using such a CN.

CN: Serverless Content-Driven Network

The straightforward way to support stateful subscriptions using CN, as already discussed in Example 2, is to "relax" them into stateless subscriptions directly supported by the network. We call this approach CN. For our running example, a simple CN such as Meghdoot can be used to support the stateless `Risk`-range subscriptions relaxed from the stateful min-`PER`-over-`Risk`-range subscriptions. Each subscriber locally maintains the content of the stateless subscription queries, and uses post-processing to derive updates to the stateful subscriptions.

The advantage of CN is its simplicity: We only need to extend the capability of the subscribers; the network substrate remains unchanged. The system does not require a central server, thereby removing a potential bottleneck. The disadvantages of CN are obvious too. All updates within the `Risk` range are sent to the subscriber, even though most of them may not affect the subscriber's range-min query result. Also, to cope with more complex updates (e.g., deletions), each subscriber must maintain all stocks within its `Risk` range, which is rather costly.

CN$^+$: CN with Additional Routing Logic

We can improve the efficiency of CN by exploiting additional information in the database state maintained for each subscription. The key observation regarding range-min subscriptions is the following (recall the notation from the beginning of Section 2.3):

> **(Subsumption property)** *If an aggregate attribute update* $\Delta(t : x, y_o \rightarrow y_n)$ *does not affect a range-min subscription with range* $[x_1, x_2] \ni x$*, then the update cannot affect any range-min subscription with range* $[x_1', x_2'] \supseteq [x_1, x_2]$.

This observation allows us to cut off forwarding of update messages early: As soon as we hit an unaffected subscription corresponding to point (x_1, x_2) in the CAN space, we can exclude the upper-left quadrant rooted at (x_1, x_2) from further forwarding. It is easy to verify the correctness of this observation. The minimum value of the aggregate attribute in the larger range $[x_1', x_2']$ is the minimum among the following three quantities: (1) the minimum in $[x_1', x_1)$, (2) the minimum in $[x_1, x_2]$, and (3) the minimum in $(x_2, x_2']$. Quantities (1) and (2) cannot change because the update falls in $[x_1, x_2]$. Thus, if the minimum in $[x_1, x_2]$ is not affected, the minimum in the larger range cannot be affected either.

We can stop forwarding even more effectively with the following observation, which is stronger and slightly more subtle:

> **(Cutoff property)** *Suppose that update* $\Delta(t : x, y_o \rightarrow y_n)$ *does not affect a range-min subscription with range* $[x_1, x_2] \ni x$ *because of another tuple* t' *with range attribute value*

$x' \in [x_1, x_2]$ *and aggregate attribute* $y' \leq \min(y_o, y_n)$. *Then the update cannot affect any range-min subscription with range* $[x'_1, x'_2] \supseteq [\min(x, x'), \max(x, x')]$.

For intuition, consider the example in Figure 2.1. Update of t_4 does not affect subscription s_2 because of stock t_2. The presence of t_2 "protects" any range-min subscription whose range includes both t_4 and t_2 (e.g., s_1) from being affected by the update of t_4. This property allows us to cut off forwarding of update messages early, instead of forwarding them all the way towards the upper-left corner of the CAN space as is done in CN.

We are now ready to present the CN$^+$ approach using our running example. For each range-min subscription, CN$^+$ maintains a stock tuple with the minimum `PER` in subscription's `Risk` range. `PER` updates (assuming for now that they are decreasing) are handled as follows. As in Meghdoot, a `PER` update to stock with `Risk` x is first routed to point (x, x) in the 2-d CAN space. The zone owner of this point tags the message with a *cutoff point* (c_1, c_2), initially set to $(-\infty, \infty)$. This message is routed upward and leftward to the zone owners in the upper-left quadrant rooted at (x, x), just as in Meghdoot, but with several differences. First, the message is not forwarded outside the rectangle spanned by (x, x) and the cutoff point. A zone owner does not need to process subscriptions outside the same rectangle because they cannot be affected. For each subscription inside the rectangle, we check its currently maintained minimum `PER` to see if it is affected. If yes, it is updated. Otherwise, we refine the cutoff point in the message based on the `Risk` value x' of the stock with the minimum `PER`: If $x' < x$, we raise c_1 to x'; if $x' > x$, we lower c_2 to x'. The cutoff property discussed earlier is the basis for this refinement.

CN$^+$ has a big performance advantage over CN because is can use the additional in-network state to cut off forwarding as early as possible. The disadvantage of CN$^+$ is its complexity. CN$^+$ pushes a significant amount of application-specific routing logic into the content-driven network layer, and the specialized routing algorithms require access to additional state including the contents of subscriptions and the database.[2] The resulting system is difficult to implement and maintain because of the lack of a clean interface separating the network from the database.

[2]We will see in Chapter 3 that handling more complex updates using CN$^+$ is trickier and requires a lot of database state to be maintained in the network.

2.3.3 Hybrid Approaches

Our goal in this section is to develop techniques that offer the same or higher level of efficiency as CN$^+$, but without complicating the network substrate with application-specific routing algorithms. To achieve this goal, we need to rethink the traditional responsibilities of servers in a publish/subscribe system, and divide the work carefully between servers and the network. We seek to maximally exploit the capability of a content-driven network within the confines of its standard interface. Recall that a content-driven network supports subscriptions defined as predicates over the content of each message. In this section, we show that with our message/subscription reformulation techniques, we can support stateful range-min subscriptions efficiently using stateless subscriptions of the form "the data rectangle in the message contains the point of interest." Such subscriptions are a standard feature in most content-driven networks, e.g., [CW01]; in particular, we show that Meghdoot can handle these subscriptions very efficiently with minimal extension. While this section focuses on range-min subscriptions, we note that message/subscription reformulation is a general mechanism; reformulation techniques for other types of subscriptions will be discussed in subsequent chapters. Reformulation itself will be described in greater detail and generality in Section 2.4.

S-CN: Server with Content-Driven Network

Under this approach, which we term S-CN, a central server maintains the database state and is responsible for generating notification messages and injecting them into a content-driven network (CN) for dissemination. Interestingly, in many cases including range-aggregation, the server does not need to know the set of subscriptions, which makes S-CN particularly attractive when subscriber anonymity is desired, or when it is expensive for a server to maintain a large, dynamic set of subscribers.

Message/Subscription Reformulation The key idea is for the server to reformulate each publish message into zero or more notification messages whose contents carry additional information derived from the current database state. This additional information effectively removes the

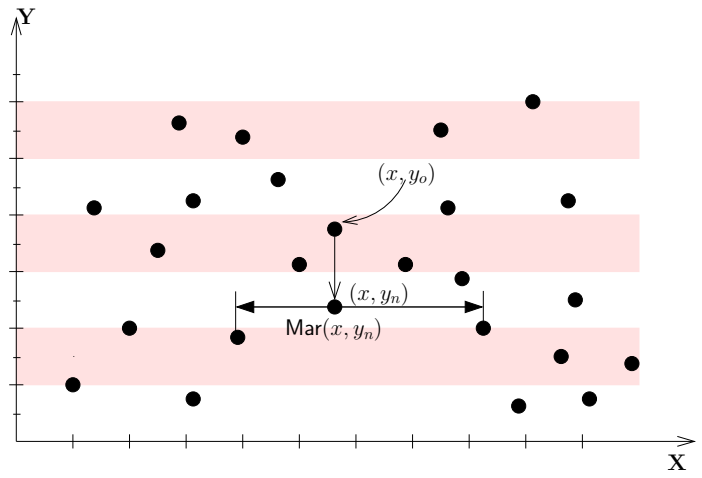


Figure 2.5: Maximum affected range (Mar).

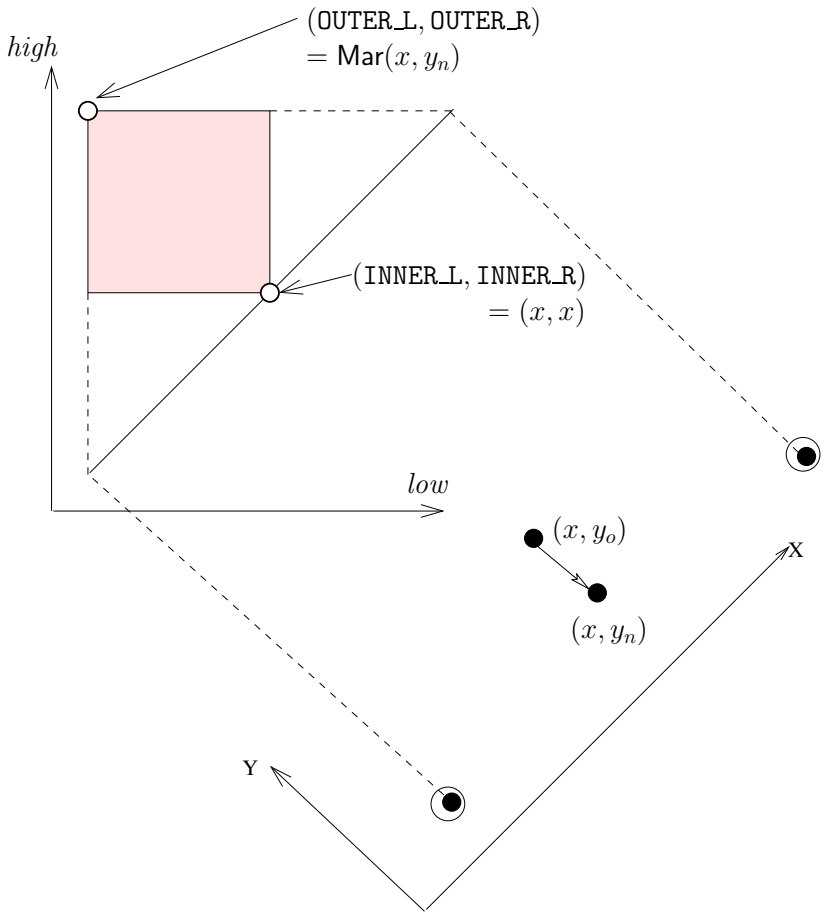


Figure 2.6: S-CN routing for a decreasing update.

dependency of stateful subscriptions on the database state. When stateful subscriptions register with the content-driven network, they are first reformulated into stateless subscriptions (without any knowledge of the database state) to work with the reformulated notification message format.

To illustrate the general reformulation mechanism, let us consider a very naive reformulation technique as a warm-up exercise. The server can simply embed the entire database state into each notification message. Doing so obviously makes all stateful subscriptions stateless, but it incurs too much overhead, and may exceed the capability of most content-driven networks as they may not support full SQL queries over the message content. How to do better than this naive technique requires non-trivial understanding of different subscription types.

Mar-Based Reformulation It turns out that for range-min subscriptions, there exists an efficient and effective reformulation based on *Mar* (for *Maximum Affected Range*), which intuitively captures an update's "extent of influence" on range-min subscriptions. Informally, using our running example, the Mar of a stock t is the maximum `Risk` range in which t has the minimum `PER` and is the only stock with this `PER`. Let a point (x, y) represent a tuple with range attribute value x and aggregate attribute value y. Figure 2.5 shows the Mar of a tuple (x, y_n), referred to as $\mathsf{Mar}(x, y_n)$, with respect to a set of points (shown as solid black dots). Basically, $\mathsf{Mar}(x, y_n)$ is an open interval between two points: the first one to the left of x and the first one to right of x, both with height less than or equal to y_n. For example, in Figure 2.1, the Mar of the t_4 update is the range $(20, 100)$. In Chapter 3, we will define Mar formally and show how to compute Mar efficiently (in time logarithmic in the size of the database).

Consider a simple decreasing update $\Delta(t : x, y_o \rightarrow y_n)$ (where $y_n < y_o$). A range-min subscription with range $[x_1, x_2]$ is affected by update Δ if and only if $x \in [x_1, x_2] \subseteq \mathsf{Mar}(x, y_n)$. The new minimum for any range-min subscription affected by this update is y_n. Based on this observation, we can perform reformulation as follows.

- **(Messages)** Each decreasing update $\Delta(t : x, y_o \rightarrow y_n)$ is reformulated into exactly one notification message

$$\langle \texttt{NEW\_MIN} : y_n, \texttt{INNER\_L} : x, \texttt{INNER\_R} : x, \texttt{OUTER\_L} : x_1, \texttt{OUTER\_R} : x_2 \rangle$$

where $(x_1, x_2) = \mathsf{Mar}(x, y_n)$, computed with respect to all the tuples other than t. This message is injected directly into CN. For example, the update $\Delta(t_4 : 40, 12 \rightarrow 4)$ in Figure 2.1 is reformulated as $\langle 4, 40, 40, 20, 100 \rangle$.

- **(Subscriptions)** Each range-min subscription over range $[x_1, x_2]$ is reformulated into a simple stateless predicate

$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R})$$

over the notification message. Upon receiving a message matching the reformulated predicate, a subscriber simply updates the minimum to NEW_MIN.

In Chapter 3, we will formally define Mar and establish its utility in range-aggregation subscription processing for all types of updates. Interestingly, with the help of Mar, we can capture all effects of any form of update on affected subscriptions succinctly and precisely, and in a way that allows the server in S-CN to encode them in the same format as the reformulated notification messages described above.

Disseminating Reformulated Messages Recall that S-CN reformulates a range-min subscription over range $[x_1, x_2]$ into the following predicate over reformulated notification messages:

$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R}).$$

We now illustrate how S-CN can disseminate messages to such subscriptions efficiently using CN without modification—specifically, a content-addressable network capable of routing to regions in CAN space, as described in Section 2.2. Recall that we can picture each subscription as a point (x_1, x_2) in a 2-d CAN space. Each reformulated notification message can be seen as specifying two opposing corners $(\texttt{INNER\_L}, \texttt{INNER\_R})$ and $(\texttt{OUTER\_L}, \texttt{OUTER\_R})$ of a rectangle in this space (as shown by the shaded region in Figure 2.6). For clarity, Figure 2.6 also shows the update and Mar (as in Figure 2.5) in the coordinate system shown at an angle of 45 degrees. This message matches precisely those subscriptions that fall within the rectangle. To support dissemination to a rectangular region, we can simply (1) start the forwarding algorithm from the lower-right corner $(\texttt{INNER\_L}, \texttt{INNER\_R})$, and (2) forward towards the upper-left, and stop forwarding once the message goes beyond the region's boundary. Note that other CN instances such as content-based networks

(e.g., [CW01]) allow subscriptions to be general predicates over message content, and therefore will also work with S-CN without any extension.

Discussion The computation of Mar-based reformulation is independent of the set of active subscriptions, which makes it easy to scale to many subscriptions (note that this is not necessarily true for all subscription types). However, some notification messages could have been avoided because certain ranges may not be covered by any active subscriptions. As a simple optimization, the server in S-CN can maintain the ranges of active subscriptions in the system, and perform a check before injecting a notification message into the network. Doing so would incur extra maintenance overhead; on the other hand, S-CN can still provide some protection of subscriber anonymity, because the server only needs to know the subscription definitions, but not who or where the subscribers are.

DS-CN: Distributing the Server in S-CN

We can replace the central server in S-CN with multiple servers that together maintain the database in a distributed manner, resulting in an approach we call DS-CN. The idea is to leverage the network substrate not only for disseminating notifications, but also for distributing the database state. In other words, we compute reformulated messages using a distributed database instead of a single server.

Consider again our running example. Using CAN as the network substrate, DS-CN maps a stock with `Risk` x to a point (x, x) on the diagonal of the CAN space. This stock would be maintained by the zone owner responsible for the corresponding point in the CAN space. In addition, each zone owner along the diagonal maintains pointers to its two immediate neighboring zone owners (left and right) along the diagonal. When a decreasing `PER` update $\delta = \Delta(t : x, y_o \rightarrow y_n)$ enters the system, DS-CN routes it to the zone owner responsible for (x, x). The zone owner then initiates two linear, distributed traversals starting from x: One traversal follows the left zone-owner pointers, scanning stocks in decreasing order of `Risk` until one (other than t) with `PER` no greater than y_n is reached; the other traversal follows the right zone-owner pointers, scanning stocks in increasing

order of `Risk` using the same stopping condition. When both traversals stop, DS-CN will have examined all stocks in $\mathsf{Mar}(x, y_n)$, which provide enough information to reformulate the update. The rest proceeds in the exact same way as S-CN.

The advantage of DS-CN over S-CN is that there is no bottleneck or single point of failure, of a central server. Published messages no longer need to rendezvous at the same server, and reformulated notification messages are now sent out from different servers. Furthermore, the standard load balancing algorithms (such as the zone splitting algorithm described in [GSAA04]) can be easily adapted to split overlay nodes that hold too much database state, thereby providing effective load balancing. The disadvantage of DS-CN is its higher implementation cost. Note that the two uses of the network substrate by DS-CN—for disseminating notifications and for distributing database state—are completely orthogonal and do not need to interfere with each other; in fact, we can use different overlay networks for the two purposes.

2.4 Reformulation: A Closer Look

We want to add support for complex subscriptions in a manner that allows quick and easy deployment, easy manageability, and good performance. In order to achieve this, in Section 2.3.3, we introduced a novel point on that spectrum of interfaces between the database and the network. This novel point uses a hybrid technique called *reformulation*, which we briefly introduced in the context of range-min subscriptions.

In general, the process of reformulation in a publish/subscribe system can be viewed as consisting of two separate components:

- As a starting point, reformulation uses a stateless *content-driven network* (CN) as the dissemination interface. CN has a well-defined and simple API interface for the database to interact with the network. Briefly, CN routes messages to destinations based on message content alone, and encompasses many dissemination schemes under its umbrella including content-based networks, CAN, distributed range-search, etc. We discussed CN in detail in Section 2.2.

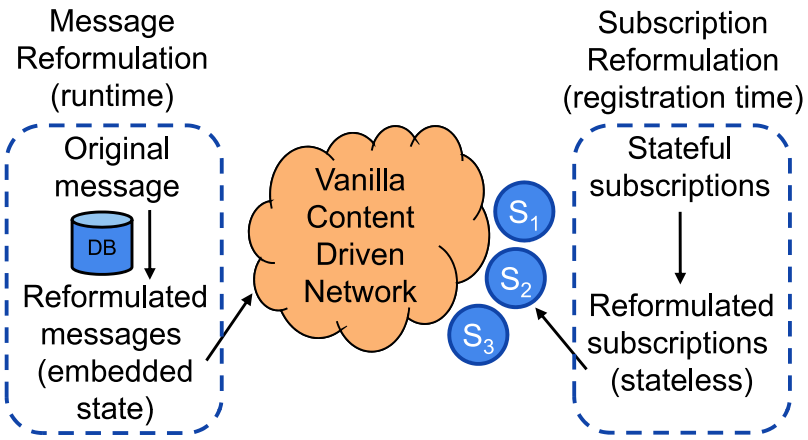


Figure 2.7: Reformulation.

- The core reformulation strategy builds around CN, by introducing specialized pre-processing of subscriptions and events at a database server in order to add support for several types (also referred to as classes, templates, or categories) of stateful subscriptions using only CN as the dissemination mechanism. Reformulation changes the role of the database server from enumerating the affected subscriptions to producing a *semantic description* of the affected subscriptions. These semantic descriptions can be efficiently group-disseminated using CN, at very low network cost. The changed database role enables new optimization opportunities that were previously unexplored. It creates the potential to develop new customized algorithms and I/O-efficient data structures that reduce server costs (potentially sublinear in the number of database tuples or subscriptions) and hence allow scaling to millions of subscriptions.

2.4.1 Overview

Reformulation is our technique for supporting stateful subscriptions over the stateless CN dissemination interface. On a high level, reformulation works as follows (see Figure 2.7). Instead of injecting an event directly into CN, we send the event to a *database server*. Events are modeled as inserts, deletes, and updates to the database. The server reformulates each event into zero or more reformulated messages with attribute-value pairs carrying additional information computed

by the server. On the other hand, each subscription is reformulated at the time of its insertion, as one or more simple stateless filters over the reformulated message schema. CN automatically routes reformulated messages to matching filters (reformulated subscriptions), thus updating their answers. The new answer to the original subscription is computed from the answers to the reformulated subscriptions. The reformulated messages, computed by the server with full access to the database and subscription definitions, carry the state necessary for processing the otherwise stateful subscriptions. We now describe each phase in the reformulation process in greater detail.

2.4.2 Event Reformulation

When an incoming event reaches the server, it triggers a change (tuple insert, delete, or update) to some table in the database at the server. We then perform event reformulation for each subscription type affected by the event.

Event reformulation for a subscription type produces a list of m (≥ 0) messages: $M_1, \ldots, M_m$. Each message contains a list of attribute-value pairs. Some pairs are attributes from the inserted, deleted, or exposed tuples—we call these the *core attributes*. Reformulation adds additional state to each message, in the form of extra attribute-value pairs—we call these the *augmented attributes*. On an incoming event, the process of reformulation at the server determines the values for the augmented attributes. The message also includes a special attribute `SUB_GROUPID`, specifying the *subscription group* (the basic unit of group processing and dissemination) for which this message is intended. Finally, a special notification type (`NOT_TYPE`) attribute may be added; this attribute indicates which *update rule* (discussed next) will be used at the affected (reformulated) subscription to update its internal state. As an optimization, if the update rule is only for the explicit deletion of some subscription state, the core attributes need to carry only a primary key for the tuple to be deleted. The process of event reformulation is illustrated in Figure 2.8.

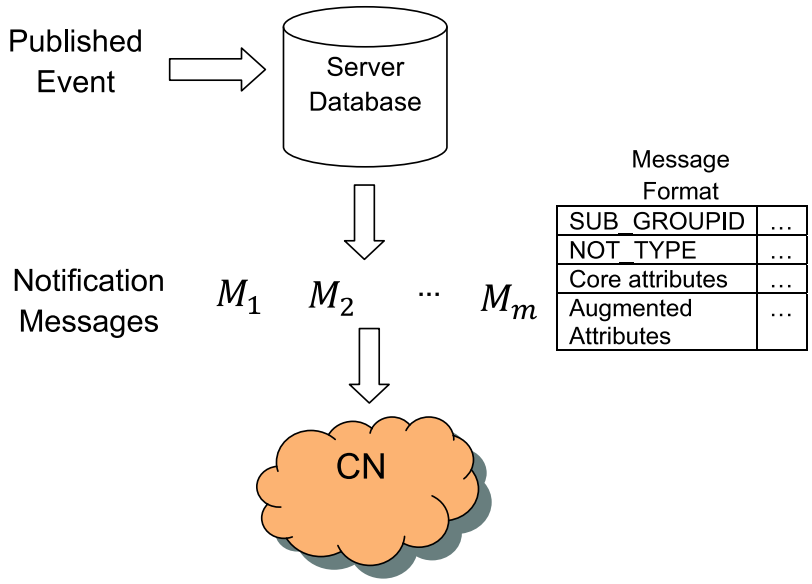


Figure 2.8: Steps in event reformulation.

2.4.3 Subscription Reformulation

When a new subscription S_i is created by a subscription client, S_i is sent to the server. The server examines the subscription and, based on the subscription type, it sends back (to the client) a non-empty set of new subscription definitions, $\mathcal{S} = \{S_i^1, \ldots, S_i^r\}$—these are called the *reformulated subscriptions*. Reformulated subscriptions are stateless, i.e., they are each simple predicates over the message content (both core and augmented attributes). In general, each reformulated subscription can belong to a different subscription group. The client registers these r reformulated subscriptions in CN in order to receive the corresponding updates.

The client maintains state associated with each of its subscriptions. Whenever an update that matches some reformulated subscription S_i^k is received, the client reads the special attributes `SUB_GROUPID` and `NOT_TYPE`, and looks up the *update rule* corresponding to that subscription group and notification type. An update rule specifies how the state associated with a subscription is updated based on the content (core attributes) of the update message. The client updates the state associated with S_i^k using the update rule.

The client then triggers the computation of a *transformation rule* that specifies how the change

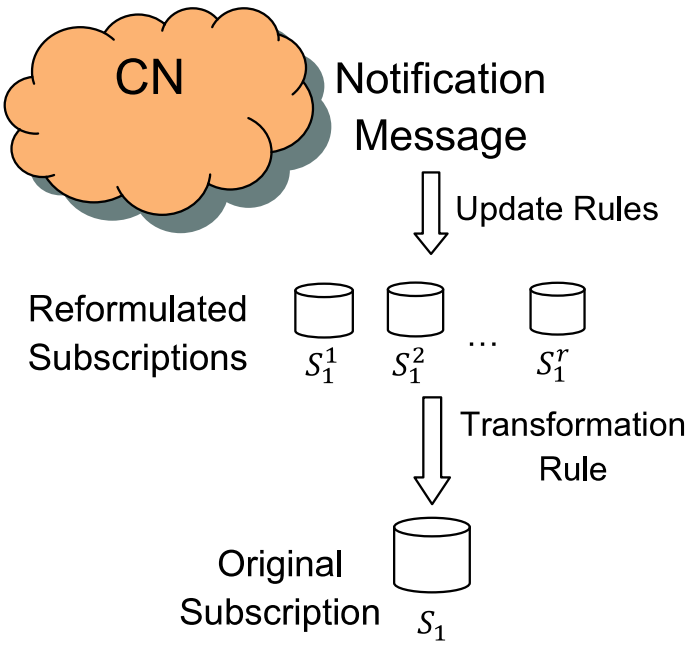


Figure 2.9: Using subscription reformulation.

to the original subscription S_i is computed from the change to S_i^k. The result of the original subscription is updated according to the rule. If there is a change to the subscription result, the user is notified of the new result. The use of subscription reformulation is illustrated in Figure 2.9.

To recap, for each subscription group, there is one update rule for each notification type, that specifies how the state associated with a subscription is updated based on the content of the update message. In addition, for each subscription type, there is one transformation rule that specifies how the change to the original subscription is computed from the change to the reformulated subscriptions. These rules can be statically retrieved by the client from the server when the client is initiated.

2.4.4 Examples

As an warm-up exercise, consider the simplest case where S_i is a stateless subscription. Event reformulation is trivial—the incoming event is directly forwarded as a message over CN. When a new subscription is created, the subscription is (trivially) rewritten as one reformulated subscription which is the original subscription itself, i.e., $S_i^1 = S_i$. The update rule updates the state with

the new tuple, while the transformation rule directly reflects the newly added state in the original subscription.

A more interesting example is the range-min subscription type introduced earlier, which asks for minimum PER within Risk ranges of interest. Our server-side event reformulation is based on the concept of *maximum affected range (Mar)*, which intuitively captures an event's "extent of influence" on subscriptions. The Mar of a stock update event is the maximum Risk range in which no other stocks have a lower PER. Suppose an event decreases the PER of a stock with $\texttt{Risk} = x$ to y. We reformulate this event into the following (single) message M_1 before injecting into CN:

$$\langle \texttt{SYMBOL} : sym, \texttt{RISK} : x, \texttt{PER} : y, \ldots,$$
$$\texttt{SUB\_GROUPID} : \text{min-per-risk-range}, \texttt{NOT\_TYPE} : t,$$
$$\texttt{IN\_L} : x, \texttt{IN\_R} : x, \texttt{OUT\_L} : x_1, \texttt{OUT\_R} : x_2 \rangle,$$

where (x_1, x_2) is the Mar of the updated stock (for events that increase PER, see Chapter 3 for details). Here, SYMBOL, RISK, and PER refer to the core attributes, SUB_GROUPID and NOT_TYPE refer to the special attributes, while IN_L, IN_R, OUT_L, and OUT_R refer to the augmented attributes.

In addition, the original subscription $S_i = \texttt{MIN(PER)}, \texttt{Risk} \in [x^i_{\min}, x^i_{\max}]$ is reformulated into one stateless reformulated subscription with group ID min-per-risk-range:

$$S^1_i = (\texttt{OUT\_L} < x^i_{\min} \leq \texttt{IN\_L}) \wedge (\texttt{IN\_R} \leq x^i_{\max} < \texttt{OUT\_R}).$$

An incoming message m that carries a new answer tuple and satisfies the predicate for S^1_i triggers a simple update rule (t) that replaces the current state for S^1_i (if any) with the new tuple which is a part of message m (the core attributes). The transformation rule for this subscription type is a trivial transformation that reports the single new tuple associated with S^1_i.

2.5 Different Approaches to Reformulation

For effective reformulation of a subscription type, we need to closely examine the characteristics of subscriptions, and how events affect subscriptions. In this dissertation, we show how reformulation can be effectively applied to a broad class of database-style subscription types. Our approach to

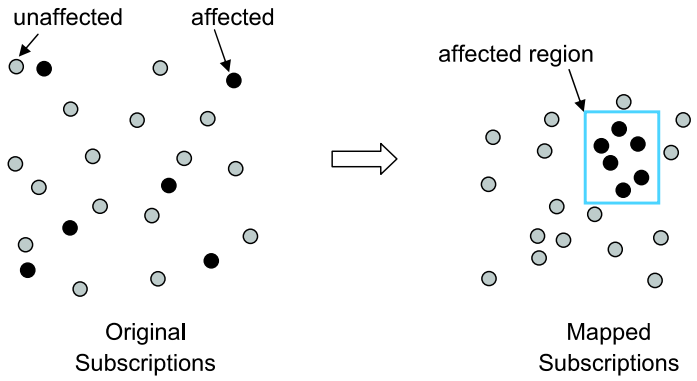


Figure 2.10: Geometric mapping for reformulation.

reformulation uses a *geometric remapping* of subscriptions, which is individually tailored to each subscription type but can give excellent performance. In this section, we describe the idea behind the geometric mapping approach, and also discuss other possible approaches to reformulation. Some of these other approaches, while having disadvantages, can potentially be independent of subscription type, thus allowing any subscription type to fit into our general reformulation framework.

2.5.1 Geometric Remapping

The geometric remapping approach towards reformulation works as follows. Consider N subscriptions of a particular subscription type which we wish to group-process and group-disseminate using reformulation. We can imagine the subscriptions to be mapped as points in some geometric space. Each incoming event exposes m (≥ 0) tuples, where each exposed tuple affects (and thus needs to be sent to) some subset of the N subscriptions. In general, the affected subscriptions could be scattered randomly in the geometric space, as shown in Figure 2.10 (left). However, by carefully choosing a mapping of subscriptions into a new geometric space, it is often possible to bring the affected subscriptions close to one another and describe them precisely using a simple region (like a rectangle) in the new mapped space, as shown in Figure 2.10 (right). The challenge in supporting a new subscription type using geometric mapping is twofold:

- Determine the correct geometric mapping of subscriptions which gives the most succinct pre-

cise representation of affected subscriptions in the form of region (or regions) in the mapped space. This often requires a careful and non-trivial analysis of event and subscription type characteristics.

- Develop efficient algorithms and data structures to compute the region (or regions), for each exposed tuple.

For example, consider again the range-min subscription type introduced earlier, where each subscription asks for the minimum PER within Risk ranges of interest. Each range-min subscription S_i over risk range $[low_i, high_i]$ is mapped to the point $(low_i, high_i)$ in a new geometric space. Suppose an event decreases the PER of a stock with Risk $= x$ to y. Let (x_1, x_2) be the Mar of the updated stock (for events that increase PER, see Chapter 3 for details). The affected region as a result of this update is a rectangle in the new geometric space, with (x_1, x_2) as the upper-left corner and (x, x) as the lower-right corner.

2.5.2 Approximate Enumeration

Enumerating the list of affected subscriptions as part of the message content can be viewed as a simple but expensive reformulation scheme. A more efficient scheme to process arbitrary subscription queries in a generic manner using reformulation is to use approximate enumeration. We register subscriptions as continuous queries at the database server. On an incoming event (modeled as insert/delete/update into the database), we can use standard continuous query processing techniques to determine, for each tuple to be disseminated, the set of subscriptions affected by that tuple. This set of subscriptions is encoded concisely at the server using a *Bloom filter* [Blo70]. A Bloom filter is a simple, space-efficient approximate representation of a set that supports membership queries. We add the Bloom filter as the augmented attribute in the message. Each CN node maintains Bloom filters for the set of subscriptions reachable on each of its outgoing links. Event matching at a broker consists of simply checking for intersection of the filter in the message, with the filter for each outgoing link. Thus, each message can reach all the affected subscriptions.

One complication with the above approach is that of false positives associated with using Bloom

filters. A subscriber can receive more notifications than necessary, and would need to perform post-processing to prune out the false positives before notifying the user. False positives can also increase the network traffic usage in the publish/subscribe middleware. Another disadvantage of this approach is that the server has to enumerate all the affected subscriptions, and thus its processing time becomes bounded by the size of the output (number of affected subscriptions) and may not scale well when many subscriptions are affected. Link matching is potentially more expensive due to the overhead of Bloom filter matching. Finally, CN has to effectively maintain and manage subscription state (in Bloom filter form) at each broker.

2.5.3 Naive State Embedding

One simple reformulation technique is to embed the entire database and per-subscription state into the message, as part of the augmented attributes portion of the message. The original stateful subscription can then be easily expressed as a 'stateless' query (which could be an arbitrarily complex set of predicates) over individual messages, and thus can be supported using CN. While correctness is ensured, this technique is usually infeasible due to the large amount of state that would need to be embedded in every message, and the potential high complexity of the subscription predicate over the message content.

2.6 Conclusions

We approach the construction of large-scale publish/subscribe systems by viewing the problem from the perspective of the interface between the database and the network. Different techniques vary in the degree of database/network cooperation; some are more suitable than others for certain types of queries and/or workloads. The tradeoffs (some of which are based on our own experiments reported in subsequent chapters) are illustrated by the following table, which compares techniques based on how they handle stateful subscriptions.

Tech-nique	Network side			Server side		Implemen-tation cost
	Traffic	MNS	State (subs)	Processing	State	
S-UN	Very high	High	None	Medium	High	Low
S-MN	Very high	Low	None	Medium	High	Low
CN	High	Low	High	None	None	Medium
CN^+	Medium	Low	Medium	None	None	High
S-CN	Low	Low	Low	Low	Low	Medium
DS-CN	Low	Low	Low	None	None	Medium

It is clear that each technique has its strengths and weaknesses. For example, although unicast does not require state at subscriptions, the update traffic can be very high. CN^+ introduces application- specific logic into the network and needs per-subscription state. S-CN and DS-CN perform well overall, with dramatic reduction in traffic at low server-side processing cost. In this dissertation, we show that simply converting a stateful subscription to a stateless one does not yield a scalable solution. Our message and subscription reformulation mechanisms are better because they can efficiently embed state information into messages. It is possible for a normal content-driven network (which can handle only stateless subscriptions) to handle several classes of stateful subscriptions efficiently: the key is to transform events into a semantic description of affected subscriptions, and subscriptions into a predicate over the semantic description.

Chapter 3

Supporting Range-Aggregation Subscriptions

Recall that we want to add support for complex stateful subscriptions in a publish/subscribe system. Towards this goal, we introduced (in Chapter 2) a spectrum of possible interfaces between the database and the network, and described reformulation as a general framework to support complex subscriptions using stateless CN. We used the simplest form of updates to range-min subscriptions as a running example in Chapter 3. In this chapter, we delve into the details of how range-aggregation can be supported by reformulation as well as by other solutions on the spectrum of interfaces. Range-aggregation subscriptions are useful in many situations where users are interesting in tracking, for example, the "best" or "worst" objects objects in ranges of their interest, e.g., stocks with the lowest price-to-earning ratios within a risk range, or lowest-priced digital cameras with at least 4.0 megapixels.

Outline We start in Section 3.1 with a general classification of updates to range-aggregation subscriptions, and then discuss several solutions to support range-aggregation. These solutions lie at different points on the spectrum of database/network interfaces. Section 3.2 discusses in detail how to support range-aggregation subscriptions using reformulation. We next describe the supporting data structures in Section 3.3. We outline extensions to support range-`DISTINCT` subscriptions as well as to handle multiple dimensions in Section 3.4. We report experimental results in Section 3.5, cover related work in Section 3.6, and conclude the chapter in Section 3.7.

3.1 Preliminaries

3.1.1 Update Classification

We give a classification of database updates based on how they affect range-aggregation subscriptions. To make our discussion concrete, recall from Example 1 the database table

$$\texttt{Stock}\,(\texttt{Symbol}, \texttt{Risk}, \texttt{PER}, \ldots)$$

and range-min subscriptions of the form

$$\texttt{SELECT MIN(PER) FROM Stock WHERE } x_1 \texttt{ <= Risk AND Risk <= } x_2.$$

We will continue to use this as the running example in this chapter. We call `Risk` the *range attribute* and `PER` the *aggregation attribute*. To simplify discussion in this section, we further restrict ourselves to updates of the aggregation attribute (`PER`) only. Insertions, deletions, and updates to other attributes require fairly straightforward extensions, details of which we defer until Section 3.3. Let $\Delta(t : x, y_o \to y_n)$ denote an update of a stock t (with risk x) that changes `PER` from y_o to y_n. This update falls into one of the following categories:

- *Ignorable updates.* These are updates that, given the current state of the database, cannot possibly affect any subscriptions. In our running example, $\Delta(t : x, y_o \to y_n)$ is ignorable if there exists another stock t' with the same risk x and a `PER` no higher than both y_o and y_n. For example, the update of t_7 in Figure 2.1 is ignorable because of t_6.

- *Non-ignorable updates.* These are updates that may affect some subscription (i.e. they are not ignorable). They are further classified into two types:

  - *Bad updates.* These are non-ignorable updates whose effects on affected subscriptions cannot be determined from the content of the update itself; additional information from the database is required. In our running example, a non-ignorable update $\Delta(t : x, y_o \to y_n)$ is bad if it might "expose" a new minimum `PER` in some risk range. The update

of t_5 in Figure 2.1 is an example of a bad update because it exposes both t_3 and t_6. The effect of a bad update cannot be inferred from the content of the update alone; additional information from the database is required.

– *Good updates.* A good update is a non-ignorable update that is not bad, i.e., its effect on any affected subscription can be determined from the content of the update itself. In our running example, a non-ignorable update $\Delta(t : x, y_o \rightarrow y_n)$ is good if no other minimum PER is exposed due to that update. The update of t_4 in Figure 2.1 is an example of a good update.

To recap, a decreasing update can be ignorable or good, whereas an increasing update can be ignorable, bad, or occasionally good. Note that this classification scheme does not take into account what subscriptions are currently in the system. Such information can be exploited for more efficiency (e.g., if there are no subscriptions, all updates are effectively ignorable), but doing so also incurs some extra overhead; we discuss this point further in Section 3.2.

3.1.2 Alternative Approaches

Before describing the reformulation-based approach in detail, we present other alternatives that form different points on the spectrum of interfaces introduced in Chapter 2.

Server with Unicast/Multicast

Assume that a single server maintains the database state and keeps track of all subscriptions. Using our group-processing techniques (to be presented in Section 3.3), we can efficiently compute all subscription updates in time sublinear in the size of the database and the number of subscriptions. Dissemination of these updates to affected subscriptions can use simple techniques like unicast (S-UN) and multicast (S-MN), as described in Section 2.3.1.

Serverless CN

We can avoid the use of servers altogether and directly use CN to support range-aggregation subscriptions, by relaxing them into stateless range subscriptions (dropping the aggregation), as described in Section 2.3.2. The subscriber performs post-processing to derive updates to the original stateful subscription. While conceptually simple, this method requires that all updates within the range are sent to the subscriber, even though most of them may be ignorable in practice. Also, to cope with bad updates, each subscriber must maintain all tuples within its range, which can be very costly.

CN$^+$: CN with Additional Routing Logic

Stateful subscriptions can be supported using additional routing logic in CN without a server. In case of range-aggregation subscriptions, we can use the subsumption and cutoff properties (discussed in Section 2.3.2) for this purpose. The basic idea is to cut off forwarding of update messages by maintaining additional state in CN (see Section 2.3.2 for details). An ignorable update is detected and stopped very early, as it will not be forwarded beyond the subscription with the smallest range containing it. CN$^+$ also attempts to cut off forwarding of non-ignorable updates as early as possible. It is possible for CN$^+$ to handle non-ignorable increasing PER updates, but the details are messy. On a high level, we can distribute the entire database state along the diagonal of the 2-d CAN space, which supports computation of any new minima exposed by bad updates. One crucial disadvantage of CN$^+$ is that it is difficult to implement and maintain because of the lack of a clean interface separating the network from the database.

3.2 Range-Aggregation using Reformulation

Our goal in this section is to develop the details of S-CN, our reformulation-based technique to handle range-aggregation subscriptions. Recall from Section 2.3.3 that in S-CN, a central server maintains the database state and is responsible for generating notification messages and injecting them into CN for dissemination. Interestingly, in case of range-aggregation, the server does not

need to know the set of subscriptions, which makes S-CN particularly attractive when subscriber anonymity is desired, or when it is expensive for a server to maintain a large, dynamic set of subscribers.

The key idea is for the server to reformulate each publish message into zero or more notification messages whose contents carry additional information derived from the current database state. This additional information effectively removes the dependency of stateful subscriptions on the database state. When stateful subscriptions register with CN, they are first reformulated into stateless subscriptions (without any knowledge of the database state) to work with the reformulated notification message format.

3.2.1 Mar-Based Reformulation

Range-min subscriptions can be efficiently and effectively reformulated based on *Mar* (for *Maximum Affected Range*), which intuitively captures an update's "extent of influence" on range-min subscriptions. Informally, using our running example, the Mar of a stock t is the maximum `Risk` range in which t has the minimum `PER` and is the only stock with this `PER`. We formally define Mar below, where a point (x, y) represents a tuple with range attribute value x and aggregate attribute value y:

Definition 1 (Maximum affected range)**.** *Mar*(x_0, y_0)*, the Mar of point* (x_0, y_0) *with respected to a set of distinct points* P*, is the maximum range* $(x_l, x_r) \ni x_0$ *for which there exists no point* $(x, y) \in P$ *such that* $x_l < x < x_r$ *and* $y \leq y_0$*. Let* Mar$(x_0, y_0) = \varnothing$ *if no such range exists; i.e.,* $\exists (x_0, y) \in P : y \leq y_0$.

The Mar of an update $\delta = \Delta(t : x, y_o \rightarrow y_n)$*, denoted* Mar$(\delta)$*, is the union of* Mar$(x, y_o)$ *and* Mar(x, y_n)*, both of which are defined with respect to the set of points representing all tuples in the relation other than* t.

For example, Figure 3.1 shows Mar(x_0, y_0) with respect to a set of points (shown as solid black dots). Basically, Mar(x_0, y_0) is an open interval between two points: the first one to the left of x_0 and the first one to right of x_0, both with height less than or equal to y_0. As another example, in

Figure 2.1, the Mar of the t_5 update is the range $(20, 100)$. We show in Section 3.3 how to compute Mar efficiently (in time logarithmic in the size of the database). The following results establish the utility of Mar in range-min subscription processing:

Theorem 1. *A range-min subscription with range $[x_1, x_2]$ is affected by an update δ if and only if $x \in [x_1, x_2] \subseteq \mathsf{Mar}(\delta)$.*

Corollary 1 (Update classification)**.** *Consider an update $\Delta(t : x, y_o \rightarrow y_n)$. (1) If $\mathsf{Mar}(t) = \varnothing$, the update is ignorable. (2) If $\mathsf{Mar}(t) \neq \varnothing$, and $\mathsf{Mar}(x, y_o) \subseteq \mathsf{Mar}(x, y_n)$ (with respect to the set of points representing all tuples other than t), then the update is good, and the new minimum for any affected range-min subscription is y_n. (3) Otherwise, the update is bad.*

Corollary 1 provides the tests for the server in S-CN to run in order to classify each incoming database update. Furthermore, this corollary leads immediately to the following reformulation techniques:

- **(Message format)** Each database update is reformulated into zero or more notification messages of the form

  $$\langle \texttt{NEW\_MIN}, \texttt{INNER\_L}, \texttt{INNER\_R}, \texttt{OUTER\_L}, \texttt{OUTER\_R} \rangle$$

  and injected into the network.

- **(Subscriptions)** Each range-min subscription over range $[x_1, x_2]$ is reformulated into a predicate

  $$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R})$$

  over the notification message. Upon receiving a message matching the reformulated predicate, a subscriber simply updates the minimum to NEW_MIN.

- **(Ignorable updates)** They are simply discarded by the server.

- **(Good updates)** Each good update $\Delta(t : x, y_o \rightarrow y_n)$ is reformulated as $\langle y_n, x, x, x_1, x_2 \rangle$, where $(x_1, x_2) = \mathsf{Mar}(x, y_n)$, computed with respect to the set of points representing all tuples other than t. For example, the good update $\Delta(t_4 : 40, 12 \rightarrow 4)$ in Figure 2.1 is reformulated as $\langle 4, 40, 40, 20, 100 \rangle$.

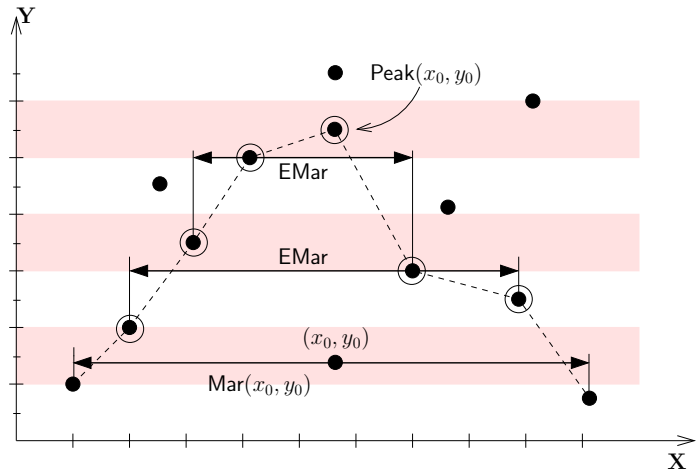


Figure 3.1: Mar and Hull.

3.2.2 Reformulating Bad Updates

As we have seen in Example 2.1, a bad update (such as the rise of t_5's PER in Figure 2.1) is tough to handle because it "exposes" different new minima for different subscriptions. Interestingly, with the help of Mar and the concept of *upper hull* introduced below, we can capture all effects of a bad update on affected subscriptions succinctly and precisely, and in a way that allows the server in S-CN to encode them in the same format as the reformulated notification messages for good updates.

Definition 2 (Upper hull)**.** *Consider point* (x_0, y_0) *and a set* P *of points. Suppose* $[x_l, x_r] = \mathit{Mar}(x_0, y_0) \neq \varnothing$. $\mathsf{Hull}(x_0, y_0)$*, the* upper hull *of point* (x_0, y_0) *with respect to* P*, is the set of points consisting of the following:*

- *The* peak*, denoted* $\mathsf{Peak}(x_0, y_0)$*, is the point* $(x_0, y) \in P$ *where* y *is the smallest possible. Let the peak be* (x_0, ∞) *if no such point exists, i.e.,* P *has no point with* X*-coordinate of* x_0*.*
- *The* left upper hull*, denoted* $\mathsf{LHull}(x_0, y_0)$*, is the set of all points* $(x', y') \in P$ *where* $x_l < x' < x_0$*, and there exists no other point* $(x, y) \in P$ *such that* $(x' \leq x < x_0) \wedge (y \leq y')$*.*
- *The* right upper hull*, denoted* $\mathsf{RHull}(x_0, y_0)$*, is the set of all points* $(x', y') \in P$ *where* $x_0 < x' < x_r$*, and there exists no other point* $(x, y) \in P$ *such that* $(x_0 < x \leq x') \wedge (y \leq y')$*.*

For example, Figure 3.1 circles the points in $\mathsf{Hull}(x_0, y_0)$. Basically, $\mathsf{Hull}(x_0, y_0)$ consists of the two "skylines" [PTFS05] that we observe by looking towards left and right from (x_0, y_0). As it turns out, each point $(x', y') \in \mathsf{Hull}(x_0, y_0)$ corresponds to a new minimum that would be exposed by the removal of (x_0, y_0). Intuitively, using $\mathsf{Mar}(x_0, y_0)$, we can capture the set of subscriptions that will have y' as their new minimum. This observation is formalized by the following theorem:

Theorem 2. *Consider a bad update $\Delta(t : x, y_o \rightarrow y_n)$. Let P be the set of points representing the set of tuples after the update has been applied. A range-min subscription with range $[x_1, x_2]$ is affected by the update if and only if there exists a point $(x', y') \in \mathsf{Hull}(x, y_o)$ (with respect to P) such that $[\min(x, x'), \max(x, x')] \subseteq [x_1, x_2] \subseteq \mathsf{EMar}(x', y')$, where $\mathsf{EMar}(x', y')$ is the* Exposed Maximum Affected Range *of (x', y') with respect to $P - \{(x', y')\}$. Furthermore, the new minimum for this affected subscription is y'.*

Note that we have used EMar instead of Mar in the above theorem. The two concepts are identical except in the special case where two points in P have the same Y-coordinate.[1] Note that the difference between EMar and Mar is rather minor and does not affect the exposition in this section. Theorem 2 provides the basis for the following technique for reformulating bad updates:

- **(Bad updates)** Given a bad update $\Delta(t : x, y_o \rightarrow y_n)$, for each point $(x', y') \in \mathsf{Hull}(x, y_o)$, the server generates a notification message $\langle y', \min(x, x'), \max(x, x'), x_1, x_2 \rangle$, where $(x_1, x_2) = \mathsf{EMar}(x', y')$ (see Theorem 2 for what point sets Hull and EMar are computed with respect to).

Both the notification message format and the behavior of reformulated subscriptions are consistent with those for good updates. The only difference is that the server generates more than one

[1] EMar differs from Mar when multiple points can share the same Y-coordinate. Specifically, $\mathsf{Mar}(x', y')$ extends to the first point (x, y) whose y is *less than or equal to* y', because for a good update, we do not want to notify a subscription whose minimum is already y. In contrast, $\mathsf{EMar}(x', y')$ in this case extends to the first point (x, y) with y *strictly less* than y'. The reason is that for a bad update, we want to send out as few messages as possible to cover all exposed minima; therefore, we want each EMar to cover as much of the exposed range as possible.

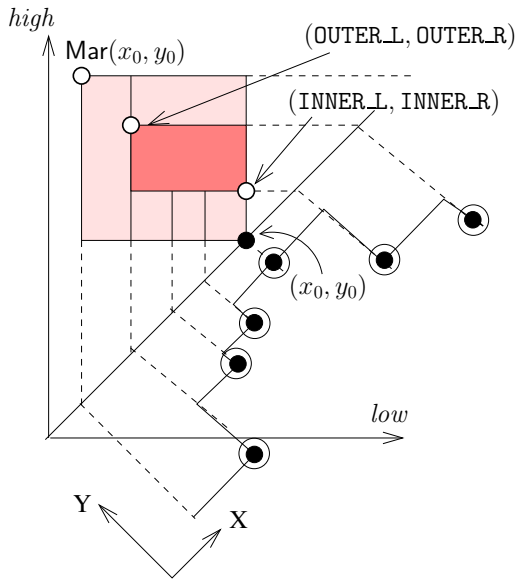


Figure 3.2: S-CN routing.

notification message per bad update. As an example, the bad update $\Delta(t_5 : 50, 5 \rightarrow 9)$ in Figure 2.1 is reformulated as 3 notification messages: $\langle 9, 50, 50, 30, 70\rangle$, $\langle 8, 30, 50, 20, 70\rangle$, and $\langle 6, 50, 70, 20, 100\rangle$.

3.2.3 Disseminating Reformulated Messages

Recall that S-CN reformulates a range-min subscription over range $[x_1, x_2]$ into the following predicate over reformulated notification messages:

$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R}).$$

Since reformulated subscriptions and messages are stateless, we can use CN directly to disseminate reformulated messages. We now illustrate how S-CN can disseminate messages to such subscriptions efficiently using CN without modification—specifically, a content-addressable network capable of routing to regions in CAN space, as described in Section 2.2. Recall that we can picture each subscription as a point (x_1, x_2) in a 2-d CAN space. Each reformulated notification message can be seen as specifying two opposing corners $(\texttt{INNER\_L}, \texttt{INNER\_R})$ and $(\texttt{OUTER\_L}, \texttt{OUTER\_R})$ of

a rectangle in this space (as shown in Figure 3.2). This message matches precisely those subscriptions that fall within the rectangle. To support dissemination to a rectangular region, we can simply (1) start the forwarding algorithm from the lower-right corner (INNER_L, INNER_R), and (2) forward towards the upper-left, and stop forwarding once the message goes beyond the region's boundary. Note that other CN instances such as content-based networks (e.g., [CW01]) allow subscriptions to be general predicates over message content, and therefore will also work with S-CN without any extension.

It is interesting to visualize the messages reformulated from good and bad updates as rectangles in the CAN space. Refer to Figure 3.2. In this figure, similar to Figure 2.6, the Mar and Hull are shown at an angle of 45 degrees. A good update is reformulated into a single rectangle, with its lower-right corner corresponding to an 0-length range containing just the update position, and its upper-left corner corresponding to the Mar of the update. A bad update is reformulated into a collection of non-overlapping rectangles, whose union is a big rectangle spanning the position and Mar of the update.

As noted earlier, we detect ignorable updates without the knowledge of the active subscriptions in the system. Thus, some non-ignorable updates may turn out to be effectively ignorable because certain ranges may not be covered by subscriptions. As a simple optimization, the server in S-CN can maintain the ranges of active subscriptions in the system, and perform a check before injecting a notification message into the network. Doing so would incur extra maintenance overhead; on the other hand, S-CN can still provide some protection of subscriber anonymity, because the server only needs to know the subscription definitions, but not who or where the subscribers are.

3.2.4 Distributed Reformulation Using DS-CN

As noted in Section 2.3.3, we can replace the central server in S-CN with multiple servers that together maintain the database in a distributed manner, resulting in an approach we call DS-CN. The idea is to leverage the network substrate not only for disseminating notifications, but also for distributing the database state. In case of range-aggregation, consider again our running range-min example. Using CAN as the network substrate, DS-CN maps a stock with Risk x to a point

(x, x) on the diagonal of the CAN space. This stock would be maintained by the zone owner responsible for the corresponding point in the CAN space. In addition, each zone owner along the diagonal maintains pointers to its two immediate neighboring zone owners (left and right) along the diagonal. When a PER update $\delta = \Delta(t : x, y_o \rightarrow y_n)$ enters the system, DS-CN routes it to the zone owner responsible for (x, x). The zone owner then initiates two linear, distributed traversals starting from x: One traversal follows the left zone-owner pointers, scanning stocks in decreasing order of Risk until one (other than t) with PER no greater than $\min(y_o, y_n)$ is reached; the other traversal follows the right zone-owner pointers, scanning stocks in increasing order of Risk using the same stopping condition. When both traversals stop, DS-CN will have examined all stocks in $\mathsf{Mar}(\delta)$, which provide enough information to reformulate the update. The rest proceeds in the exact same way as S-CN.

The advantage of DS-CN over S-CN is that there is no bottleneck of a central server. Publish messages no longer need to rendezvous at the same server, and reformulated notification messages are now sent out from different servers. Furthermore, the standard zone-splitting algorithm (such as the one described in [GSAA04]) can be easily adapted to split diagonal zones that hold too much database state, thereby proving effective load balancing. Note that the two uses of the network substrate by DS-CN—for disseminating notifications and for distributing database state—are completely orthogonal and do not need to interfere with each other; in fact, we can use different overlay networks for the two purposes.

3.3 Server Data Structures for Range-Min[2]

3.3.1 Data Structures for Range-Min in S-CN

Computing Mar with A2B-Tree

We begin this section by discussing the data structure and algorithm for computing Mar, which is a primitive used by S-CN for both update classification and message reformulation. In Section 3.3.2,

[2]This section summarizes joint work described in detail in [CXY05].

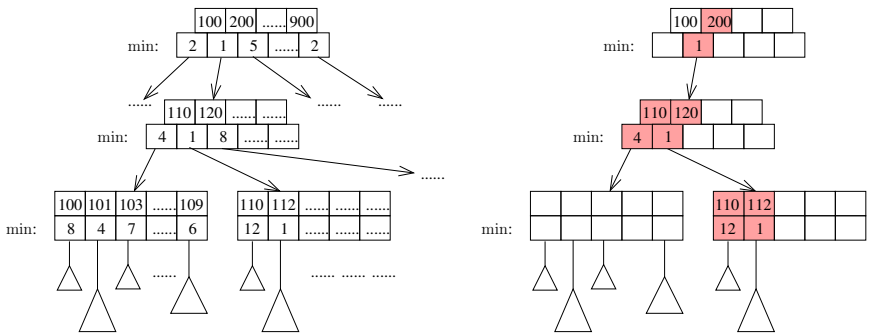


Figure 3.3: Example of an A2B-tree.

we will also show how to apply the same techniques in database-centric approaches to compute notification messages. In the following, we denote the range attribute by X and the aggregate attribute by Y.

A straightforward way to compute Mar is to use a B-tree on X. Given an update $\delta = \Delta(t : x, y_o \rightarrow y_n)$, we look up the tuple(s) with $X = x$ in the B-tree. Starting from these tuples, we scan the index leaves left and right in a manner similar to DS-CN (but on one server). The number of I/Os incurred is $O(\log_B N + L/B)$, where N is the size of the database table, L is the number of tuples in $\mathsf{Mar}(\delta)$, and B is the block size of the B-tree. However, this method does not scale with large affected ranges.

We propose *A2B-tree* (*augmented* 2*-tier B-tree*) for computing Mar. The upper tier is a B-tree on X. Each leaf index entry of this upper-tier B-tree points to a lower-tier B-tree indexing all tuples with the same X value, using Y as the index key. Furthermore, each upper-tier index entry (which points to either a child in the upper tier or the root of lower-tier B-tree) is augmented with an extra $\min$ field maintaining the minimum Y value found in the subtree rooted at this index entry. An example A2B-tree is shown in the left part of Figure 3.3. The space taken by this index is $O(N/B)$, linear in the size of the table. Both the height of the upper tier and the height of the lower tier are bounded by $O(\log_B N)$, so the combined height is also $O(\log_B N)$.

A2B-tree supports lookups and updates in $O(\log_B N)$ I/Os. Lookup follows the standard B-tree procedure, first using X as the search key through the upper tier, and then using Y as the search key through the lower tier. Insertion and deletion extend the standard B-tree procedures with

maintenance of $\min$ fields.

Our A2B-tree can be easily maintained by standard B-tree maintenance procedures with slight modifications to adjust $\min$ fields accordingly. For example, when inserting a tuple with $Y = y$, for each upper-tier index entry on the path from the root to the lower-tier B-tree containing the insertion point, we update $\min$ to y if it is currently greater than y. Deletion is slightly trickier than insert because it may remove the tuple who is supplying the $\min$ value for some upper-tier index entry. We proceed in a bottom-up manner, first updating $\min$ of the upper-tier leaf index entry (the new value can be easily obtained from the leftmost leaf of the corresponding lower-tier B-tree), and then updating each entry long the path from left to root until the root, or an entry which does not need to be updated, whichever comes earlier. Similarly, in split and merge, $\min$ filed should also be maintained. For each entry, new $\min$ can be computed in constant time from its children and there are $O(\log n)$ entries need to be modified, therefore, maintaining the auxiliary data does not increase the time complexity of standard B-tree maintenance procedures. Therefore, the I/O cost is bounded by $O(\log_B N)$ in maintenance.

Conceptually, we use the following simple example to show how Mar is computed. Assume that we want to compute the right end of $\mathsf{Mar}(x_0, y_0)$ given a point $(x_0, y_0) = (102, 3)$ in the left part of Figure 3.3. At top level, we first identify the subtree containing $x_0 = 102$, which is associated with key 200. Since the min field of that entry is 1, smaller than $y_0 = 3$, we need to go one level down by following the link to subtree. At the second level, we first identify the subtree containing x_0 again (subtree with key 110 in example), however the min field of this entry is 4, greater than y_0, hence we are sure we cannot find right end of Mar in this subtree since every point in which has greater Y value than y_0. Thus we move rightward to locate the first entry which min field no greater than y_0, which is subtree with key 120 and min field 1, and we go further down to the third level, which is leaf level of our A2B-tree. We repeat the traverse process above to find right end of Mar is $(112, 1)$. The shaded squares in right part of Figure 3.3 are the entries we visit during the process. We simply return ∞ if we do not find any point in the above process.

Algorithm 1 for $\textsc{FindR}_<(\mathcal{T}, x, y)$ queries right boundary of Mar of a given tuple (x, y) on an augmented B-tree $\mathcal{T}$, assuming domain of x is $(-\infty, +\infty)$ and each node has B entries at most.

Starting from root of $\mathcal{T}$, it traverses downward along the path from root to where x is located, until reaching an entry whose $\min$ field is greater or equal to y. Then it calls subroutine SEARCHR$_<$ (Algorithm 2) to continue the search rightward. Note that in order to compute FINDR$_\leq(\mathcal{T}, x, y)$, we just need to replace $\leq$ with $<$ in Line 8 of Algorithm 1 and replace FINDR$_<$ and SEARCHR$_<$ with FINDR$_\leq$ and SEARCHR$_\leq$ respectively.

Since at every level of the upper-tier of A2B-tree, the algorithm visits two nodes at most, its time complexity is $O(\log_B N)$ (bounded by twice the tree height). Computation of Mar serves as a building block for the subsequent sections, where we will introduce efficient computation of updates to a large number of range-aggregation subscriptions given an incoming event.

Computing Hull and EMar

S-CN needs to compute Hull and EMar in order to reformulate bad updates. Computation of Hull(x_0, y_0) begins with identifying the peak, which is easily located by looking up x_0 in the A2B-tree $\mathcal{T}$. Starting from the peak (x_0, y_p), we call FINDR$_<(\mathcal{T}, x_0, y_p, \varnothing)$, which returns the X-coordinate of the first point (x_1, y_1) in RHull; y_1 can be readily obtained from the $\min$ field of the leaf index entry for x_1. Then, for $i \geq 1$, we repeatedly call FINDR$_<(\mathcal{T}, x_i, y_i, \varnothing)$ to locate the next point (x_{i+1}, y_{i+1}) in RHull until $y_{i+1} \leq y_0$. The last point is discarded. LHull(x_0, y_0) is computed analogously. Since the cost of each FINDR or FINDL is $O(log_B N)$, the total cost of computing Hull(x_0, y_0) is $O(k \log_B N)$, where k is the number of points in the Hull. All index nodes visited in the process belong to the minimum spanning tree containing all points in Hull; we can further ensure that we never visit the same node more than once, although this optimization does not change the asymptotic complexity.

Recall from Section 3.2 that S-CN reformulates a bad update using EMar for each point in Hull. While we could compute each EMar individually with a procedure similar to Mar computation, it is much more efficient to use the already computed Hull. The key (illustrated in Figure 3.1) is that for Peak, EMar begins at the rightmost point in LHull and ends at the leftmost point in RHull; for each point (x_i, y_i) in LHull, EMar(x_i, y_i) begins at its left neighbor in LHull, and ends at the first point (x', y') in RHull with $y' < y_i$; for each point (x_i, y_i) in RHull, EMar(x_i, y_i) begins at the last

Algorithm 1: FINDR algorithm.

```
1  FINDR_<(T, x, y) begin
2      m ← +∞, node ← T.root;
3      if node = ∅ then                                   // tree is empty
4          return m;
5      i ← 1;
       // locate index i of subtree by key x
6      while i ≤ B and node.key[i] < x do
7          i ← i + 1;
8      if y ≤ node.min[i] then
9          m_r2 ← SEARCHR_<(node, y, i + 1);
10     else
11         if leaf[node] then                             // node is a leaf
12             m ← node.key[i];
13         else
               // node is an internal node
14             n ← FINDR_<(node.child[i], x, y);
15             if n ≠ +∞ then m ← n;
16             else m ← SEARCHR_<(node, y, i + 1);
17     return m;
18 end
```

Algorithm 2: SEARCHR algorithm.

```
1  SEARCHR_<(node, y, index) begin
       // Replace the second ≤ with < in Line 3
       // to compute SEARCHR_≤(T, x, y)
2      i ← index;
3      while i ≤ B and y ≤ node.min[i] do
4          i ← i + 1;
5      if i = B + 1 then
6          m ← +∞;
7      else
8          if leaf[node] then                          // node is a leaf
9              m ← node.key[i];
10         else
11             m ← SEARCHR_<(node.child[i], y, 0);
12     return m;
13 end
```

point (x', y') in LHull with $y' \leq yi$, and ends at (x_i, y_i)'s right neighbor in RHull.[3] We exploit the fact that LHull and RHull computation naturally produces points in sorted order. Using a merge-like procedure on LHull and RHull, we can compute EMar for all points in Hull in time $O(k)$, where k is the number of points in Hull, which is also the number of reformulated messages sent out by S-CN for the bad update.

Handling Insertion, Deletion, and Other Updates

So far, for simplicity of presentation, we have only discussed updates of the aggregate attribute. We now briefly present how to support other types of modifications. For insertion and deletion, all results in Section 3.2 continue to hold if we view an insertion as an update $\Delta(t : x, \infty \rightarrow y_n)$, and a deletion as $\Delta(t : x, y_o \rightarrow \infty)$. These tuples involving ∞ are of course never stored explicitly; we simply assume their existence for convenience. Updates that change neither X nor Y are ignored. For a complex update $\Delta(t : x_o \rightarrow x_n, y_o \rightarrow y_n)$ that change both X and Y, we could treat it as a deletion of (x_o, y_o) followed by an insertion of (x_n, y_n). With this simple method, however, it is possible for a subscription to receive two notification messages (one due to deletion and one due to insertion). It is easy to ensure that each subscription gets at most one notification for each incoming event, by consolidating the effects of a complex update. This approach has complexity $O(k \log_B N)$, where k is the minimum number of unique notification massages required for a complex update.

3.3.2 Data Structures for Range-Min in S-UN and S-MN

We briefly discuss how to use the A2B-tree and concepts of Mar and Hull to compute unicast and multicast notification messages efficiently for S-UN and S-MN. Given an update, a naive method is to examine every subscription in turn and compute how the update affects it; the A2B-tree can

[3]Note the slight asymmetry in defining EMar for LHull and RHull points ($y' < y_i$ vs. $y' \leq y_i$). In the special case where a point in LHull has the same Y-coordinate as a point in RHull, we need this asymmetry to ensure the property that every subscription affected by a bad update receives exactly one notification message (i.e., no redundant or missing notifications).

be used to recompute in $O(\log_B N)$ time the new minimum in any range when the old minimum is lowered or deleted. This method takes $O(M \log_B N)$ time, where M is the number of subscriptions; it does not scale to a large number of subscriptions. We can do much better by leveraging the techniques in Section 3.3.1 for S-CN. Using the same algorithms and an A2B-tree for the database table, we compute reformulated messages for each incoming update as in S-CN. Instead of disseminating these messages, however, we treat each message as a query identifying the set of affected subscriptions. As in Figure 2.4, we view the range-min subscriptions as points in a 2-d space, and store them using a 2-d index structure that supports rectangular queries. An external-memory kd-tree, for example, would support such a query in $O(\sqrt{M/B} + J/B)$ time, where J is the number of subscriptions in the query rectangle. S-UN generates one unicast message for each such subscription; S-MN generates one multicast message that encodes this set of subscriptions. Overall, for an update with k reformulated messages in S-CN affecting a total of A out of M subscriptions, the server-side processing costs incurred by S-UN and S-MN are $O(k \log_B N + k\sqrt{M/B} + A/B)$, compared with $O(k \log_B N)$ for S-CN.

3.4 Discussion

3.4.1 Other Range-Aggregation, Range-DISTINCT Subscriptions

In this section, we briefly discuss how to handle several other classes of subscriptions using our general message/subscription reformulation mechanism. More subscription types (including select-joins and selects with value-based notification conditions) are considered in subsequent chapters.

Range-max subscriptions can be handled by the same techniques as range-min. Range-count, -sum, and -average subscriptions are easier to handle: We simply reformulate them into range subscriptions without aggregation; publish messages do not need to be reformulated (though obviously irrelevant updates can be ignored, e.g., those updating neither range nor aggregation attributes). Unlike range-min and range-max, relaxing these range-aggregation subscriptions would not result in excessive traffic, because relevant updates that fall within a subscription range generally do affect

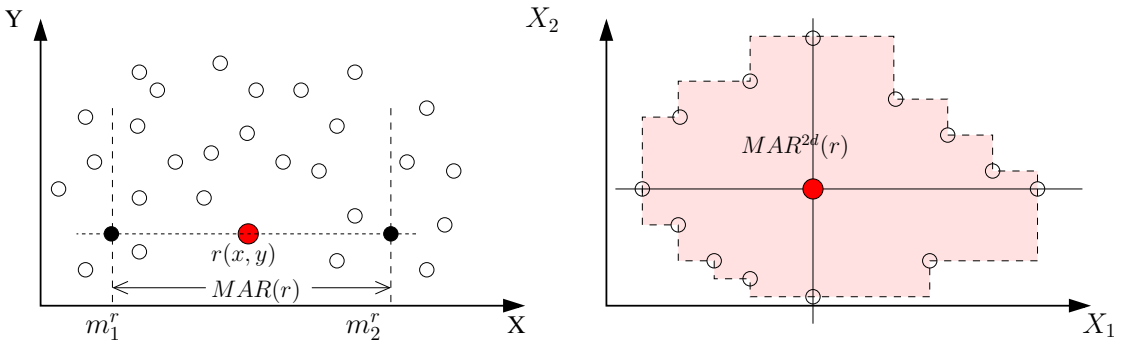


Figure 3.4: (a) Handling range-DISTINCT. (b) Generalizing Mar to 2-d.

the subscription.

A range-DISTINCT subscription tracks the set of distinct values of an attribute Y for tuples whose range attribute X fall within some range. Simply relaxing this subscription into a range subscription may generate a lot of unnecessary traffic if there are many duplicates. In this case, we can extend the concept of Mar as follows: Mar of an insertion (or deletion) is the maximum X range that contains the insertion (or deletion) point and no other tuples with the same Y value, as shown in left part of Figure 3.4. An insertion (or deletion) is reformulated into a message containing X and Y values and the Mar, if it is not empty. A range-DISTINCT subscription is reformulated into a stateless selection subscription that checks if the subscription range contains the X value and is also contained by the Mar.

In a publish/subscribe system that uses a single network substrate to support more than one types of subscriptions, we use an additional TYPE field in reformulated messages to distinguish those intended for different types of subscriptions. A reformulated subscription would also include an extra condition that selects notification messages with the appropriate TYPE value.

3.4.2 Extending to Higher Dimensions

We have extended our techniques for 1-d range aggregation/DISTINCT subscriptions to the 2-d case, where each subscription can specify orthogonal range conditions for two attributes. The shaded area in right part of Figure 3.4 is an example of Mar^{2d} for a tuple r. The contour of shaded area is determined by a set of tuples in 2d range selection space X_1, X_2 such that for each of them r',

$r'.y \leq r < y$ and there does not exist a tuple r'' in the shaded area such that $r''.y \leq r.y$. It is not hard to prove that a subscription will be affected by update of r if and only if it is stabbed by r and its orthogonal range selection condition is fully contained within Mar^{2d}. We leave as future work the data structure and algorithms for computing Mar^{2d} and Hull in two dimensions.

There are two problems with a direct extension of our scheme to higher dimensions: (1) the data structures for computing Mar^{2d} as less efficient compared to the 1-d case (primarily due to the complexity of skyline computation), and (2) the size of Mar^{2d} can be quite large—up to the size of the database in the worst case, as opposed to the one-dimensional case where Mar has constant size. One practical solution to handle the higher complexity is to approximate the Mar as a bounding rectangle, which trades off description complexity by allowing false positives.

3.5 Evaluation

3.5.1 Implementation Details

On the server side, we implemented the algorithms computing Mar and Hull, and the processing techniques introduced in Section 3.3. To compare with S-CN, we also implemented the query processing component for unicast and message grouping component for multicast. The server module supports well-defined network interfaces to a regular network for unicast and multicast, and CAN for S-CN. On the network side, we have implemented a network simulator for a large-scale publish/subscribe system. The first phase of network simulation generates application-level routing details that are used by a second phase which can accept any topology generated using INET [CGJ+02], an Internet-like network topology generator. This phase performs a link-level simulation (timing is not simulated) of the network topology. We support a number of dissemination styles, as discussed next.

Unicast from a centralized server. First, the server determines the set of subscribers affected by an update. The network simulator sends a single hop unicast message to each subscriber, carrying the new answer to that subscription. The route follows the shortest path over the underlying IP

substrate from the server to the destination.

Multicast from a centralized server. For each exposed answer, the server determines the set of destinations that need to receive a multicast message with that answer.

A message containing this new answer tuple has to be multicast to the set of destinations. The network simulator uses this application-level data to perform multicast as described next. Given a set of N possible destinations, it would take 2^N groups to be able to directly multicast to any subset of recipients. We use application-level multicast in our simulator, as IP multicast is not widely supported and has severe limitations in terms of number of groups. We implement an efficient hierarchical multicast approach that limits the number of groups that any single node needs to be aware of.

The hierarchical multicast that we have designed and implemented is inspired by the techniques used in *NICE* [BBK02] and works as follows. Consider a network with N nodes. In addition to its IP address, each node can be identified by its position within the network, described as a set of coordinates in a geometric space such as the one proposed by *Global Network Positioning* (*GNP*) [NZ02]. GNP assigns coordinates to nodes such that their geometric distances in the GNP space approximate their actual network distances. We then use a geographic clustering algorithm such as k-means to form a tree of nodes as follows. We define a limit c on the number of children at any node in the tree. Clustering is performed on a set of n nodes to give $min(c, n/c)$ sub-clusters of nodes. Each sub-cluster has a leader, and the leader of the original cluster is the parent of these sub-cluster leaders in the tree. The tree is rooted at the server which acts as the leader of the highest level cluster consisting of all nodes in the network. Each cluster C with leader L_C is further clustered in the same manner and L_C is the parent of the leaders of each of these sub-clusters. Clustering is no longer performed on a cluster if it has no more than c nodes. This gives a tree rooted at the server, with each non-leaf node having at most c children, and the tree has a total of N leaves. Each leader L with k children forms a multicast group for each of the 2^k possible subsets of the k children. It maintains a routing table that provides, for each multicast group, the application-level multicast tree that is constructed rooted at L and reaching all the members of that multicast group. The application-level multicast tree can be constructed in a number of ways—we use a greedy offline

algorithm to construct a bottleneck bandwidth tree; this was used in Bullet [KRAV03] as a good offline technique to compare with dynamic application-level multicast techniques. Note that the number of group IDs at any node is bounded by 2^c and hence, by limiting c we can ensure that no node needs to be aware of a large number of multicast groups.

When a node L receives a message along with a set S of recipient nodes for that message, it can compute the subset of its children that need to receive the message. All it needs to compute this is a bitmap of all the nodes, formed by a left-to-right ordering of the nodes in the tree under node L. At every level, the nodes in each sub-cluster form a contiguous chunk in the bitmap. By looking at the bit positions that need to be reached in its chunk of the bitmap, a node can determine the subset of its children to which the message needs to be forwarded. It can then look up the multicast group ID for that subset and forward the message along the application-level multicast tree described earlier. This process is repeated until the message reaches all its intended recipients. Note that the message reaching a node N needs to encode the exact set of destination subscribers located below N as this is needed to determine the set of affected children at that node.

CN-based routing with range predicates, with Mar, and with additional state (no server). In case of techniques that use CN, we use a CAN simulator based on Meghdoot [GSAA04], with added support for routing to fixed regions. The CAN is built by starting with a single node responsible for a single zone in the network. New zones are created by splitting existing zones with large number of subscriptions, or replicating zones with high traffic passing through them. A new zone is created each time a new node is inserted at random points during the bootstrap phase of the simulation. Once we reach a stable CAN configuration with all nodes inserted into the overlay, we simulate the injection of a large-number of events as per the workload, and track the paths followed by each message during dissemination. These paths are used by the link-level network simulator, which generates both overlay-level and IP-level statistics.

We use this simulator in order to evaluate the CN and S-CN approaches, without adding specialized logic to handle our stateful queries. The simulator is augmented with our link-level simulator for more accurate routing statistics. In order to support the sophisticated techniques used in CN$^+$ (such as early cutoff) and DS-CN (such as diagonal two-way routing for Mar computation), we

implemented appropriate extensions to the simulator.

3.5.2 Evaluation Metrics

We use both server- and network-side metrics for evaluation. On the server-side, we track processing time, which is measured as the period between the time at which an update arrives at the server and the time at which the server completes generation of all outgoing messages for dissemination. On the network-side, we track, for each event: (1) *Number of overlay message hops*: This is the total number of messages sent between overlay nodes, in order to process and disseminate that event. (2) *Number of IP message hops*: This the number of hops over IP-level links, during dissemination of an event. An overlay hop may traverse a number of IP-level links on its path. (3) *Network traffic*: We define network traffic as the total number of bytes that need to be transferred between overlay nodes during dissemination. (4) *Maximum node stress (MNS)*: We define node stress as the number of overlay messages originating from a node. MNS for an event is the highest node stress among all nodes while processing that event.

3.5.3 Workload

We use normal distributions to derive 1-d range-min subscriptions. To model a *hot* range which interests more subscribers, the position of the subscription interval is normally distributed around center of domain of local selection attribute. The interval length also follows a normal distribution. This subscription workload is used in association with both synthetic and real update workloads.

Our synthetic update workload consists initially of a database that contains between $10,000$ and $100,000$ tuples uniformly distributed in the domain. We generate $200,000$ events (long enough to reach stable measurements), each being an update of the aggregate attribute `PER`, and collect the measurements of each update. All experimental parameters are summarized in Table 3.1. We vary one or more of these parameters to perform experiments with varying database size, number of subscriptions, percentage of ignorable updates, and the average number of subscriptions affected.

We update a tuple by increasing or decreasing its output attribute using a random walk model.

parameter	value
number of overlay nodes	1000
number of physical nodes	$20,000$
domain of range select attribute	$[0, 10, 000]$
domain of aggregate attribute	$[0, 10, 000]$
number of subscriptions	$100k - 1M$
midpoint of selection range	$N(5000, 1500)$
length of selection range	$N(1000, 1000)$
number of tuples in initial DB	$10k - 100k$
number of simulation events	$200,000$
percentage of spiked events	0.5%

Table 3.1: Summary of all parameters used in experiments.

The step depends on the current value and updates are independent of each other. Each random variable (tuple) in this model is actually an irreducible, finite, and aperiodic Markov chain; hence there exists a stationary distribution for the value of each tuple. Consequently, update statistics such as the number of subscriptions affected by an update will be stationary over time. Our simulation is long enough to ensure convergence. To make the workload more realistic, we also introduce a small percentage of *spikes*. A spike is an update where the PER drops suddenly (affecting a large number of subscriptions) and then bounces back to its old value. We also use a real update workload based on stock data from Yahoo! Finance [Yah]. More details are provided in Section 3.5.5.

3.5.4 Experimental Setup

We perform detailed link-level simulation of a $20,000$-node INET topology. Of these, 1000 nodes are chosen as the end nodes participating in an overlay network. Events are generated by publishers assumed to be randomly scattered throughout the network.

We assume that publishers are distributed throughout the network. We do not model the mes-

sage hop from the publisher to the server/event point, because this cost would be incurred in all systems. If the publisher is not a part of the CAN, it could contact some peer as an entry point into the CAN. Similarly, subscriptions could be distributed throughout the network, but each subscription chooses a peer in the overlay network as its gateway to the CAN. We use the following technique to choose a gateway in our experiments. In CAN-based techniques, the zone owner is chosen as the gateway for all subscriptions that map to that zone. The same mapping is used in all the compared approaches. We assume that subscriptions reside at the gateway and do not model the propagation of messages from the gateway to the end subscriber. In case of multicast, the network-side metrics for each group ID at each node in the hierarchy are precomputed to speed up the simulation. In addition, c is fixed at 10 so that no more than 1024 multicast trees need to be stored at any node in the system.

3.5.5 Experiments and Results

Demonstration of scalability

In this set of experiments, we compare the techniques in terms of their ability to scale to large numbers of tuples and subscriptions. On the server side, we compare the average processing time per update for S-CN, unicast and multicast (recall that CN and DS-CN do not have a central server). On the network side, we compare the average network traffic (bytes) generated per event. All updates are non-ignorable and the percentage of spikes is 0.5%.

We first vary the database size from $10,000$ to $100,000$ tuples, for $500,000$ subscriptions. Figures 3.5 and 3.6 show the server- and network-side costs respectively. On the network side, CN, S-CN, and DS-CN perform much better than unicast and multicast, and the average network traffic is independent of database size. Note that we have shown network traffic in log scale on the y-axis because of the order of magnitude difference between the various approaches. Among these three approaches, DS-CN performs the best in terms of traffic as the diagonal traversal usually takes very few hops. S-CN is next, the main factor for performing worse than DS-CN is the cost of routing from the server to the event point. This is followed by CN. Among central server approaches,

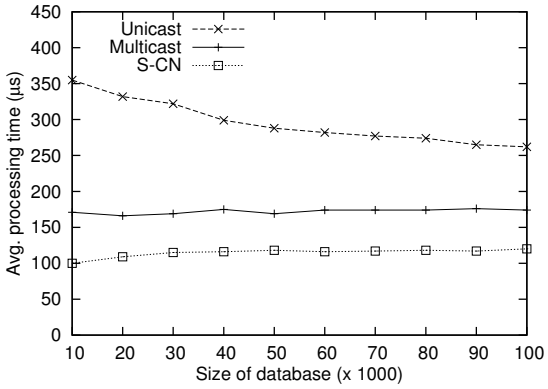


Figure 3.5: Avg. processing time; increasing database size.

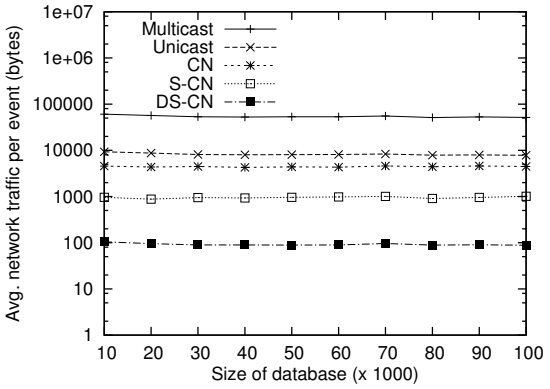


Figure 3.6: Avg. network traffic; increasing database size.

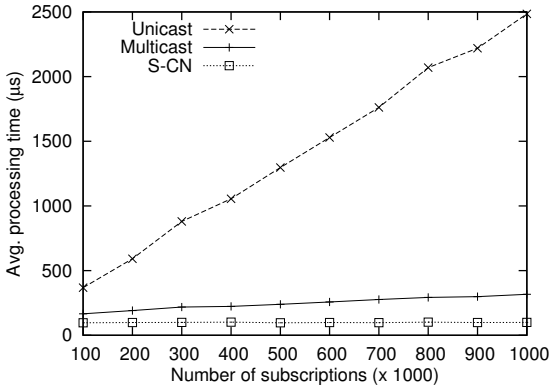


Figure 3.7: Avg. processing time; increasing number of subscriptions.

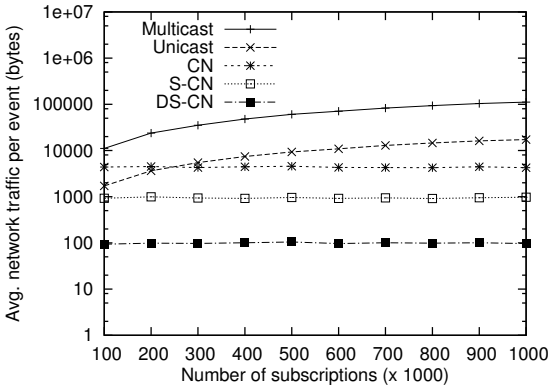


Figure 3.8: Avg. network traffic; increasing number of subscriptions.

S-CN achieves the lowest processing time, and its processing time increases only by 20% when the database size increases by a factor of 10. The processing time of multicast is roughly 50-70% higher than S-CN due to the cost of identifying all affected subscriptions. The trend for multicast is flat because the slight increase in processing time over increasing database size is compensated by a decreasing number of affected subscriptions: a larger database gives a subscription more resistance against update. However, the network-side performance of multicast is the worst because the set of affected subscriptions needs to be encoded in each message, even though the number of multicast messages may be small. Unicast performs badly on both server- and network-side due to the high cost of assembling and disseminating outgoing messages for each affected subscription. The processing time of unicast also benefits from a larger database.

db size	$20K$	$40K$	$60K$	$80K$	$100K$
S-CN	2.10	2.15	2.18	2.20	2.22
Unicast	726	671	678	655	654

Table 3.2: Number of outgoing messages from server, varying database size.

# subs.	$200K$	$400K$	$600K$	$800K$	$1M$
S-CN	2.08	2.08	2.08	2.08	2.08
Unicast	303	616	909	1218	1441

Table 3.3: Number of outgoing messages from server, varying num. of subscriptions.

Next, we keep the database size at $10,000$ tuples and vary the number of subscriptions from $100,000$ to 1 million. Figures 3.7 and 3.8 show scalability on the server and network side respectively. S-CN is completely independent of subscriptions; thus, both its processing time and network traffic is approximately constant. The three approaches of CN, S-CN, and DS-CN perform well with traffic never rising above $1kB$. All these approaches are independent of the number of subscriptions, which makes them scalable. The processing time of multicast increases over number of subscriptions. Unicast also increases but it performs much worse due to the overhead of memory allocation for constructing objects corresponding to outgoing messages (there are many outgoing messages for unicast). Even if we can optimize it using bulk memory allocation, it cannot beat multicast, which is worse than S-CN. On the network side, multicast is good in terms of number of overlay hops (not shown) but it performs badly in terms of total traffic generated.

Tables 3.2 and 3.3 compare the number of outgoing messages (from server) of S-CN and unicast for the two sets of experiments. In all cases, S-CN saves more than 99% of outgoing messages. We also evaluated CN$^+$ for the portion of workload with decreasing updates. We found that early stopping is effective and results in traffic reduction of more than 98% compared to CN. Early stopping in CN$^+$ (for decreasing updates) was around 2% less effective in terms of traffic than the Mar-based stop conditions in S-CN. However, handling increasing updates in CN$^+$ is complex

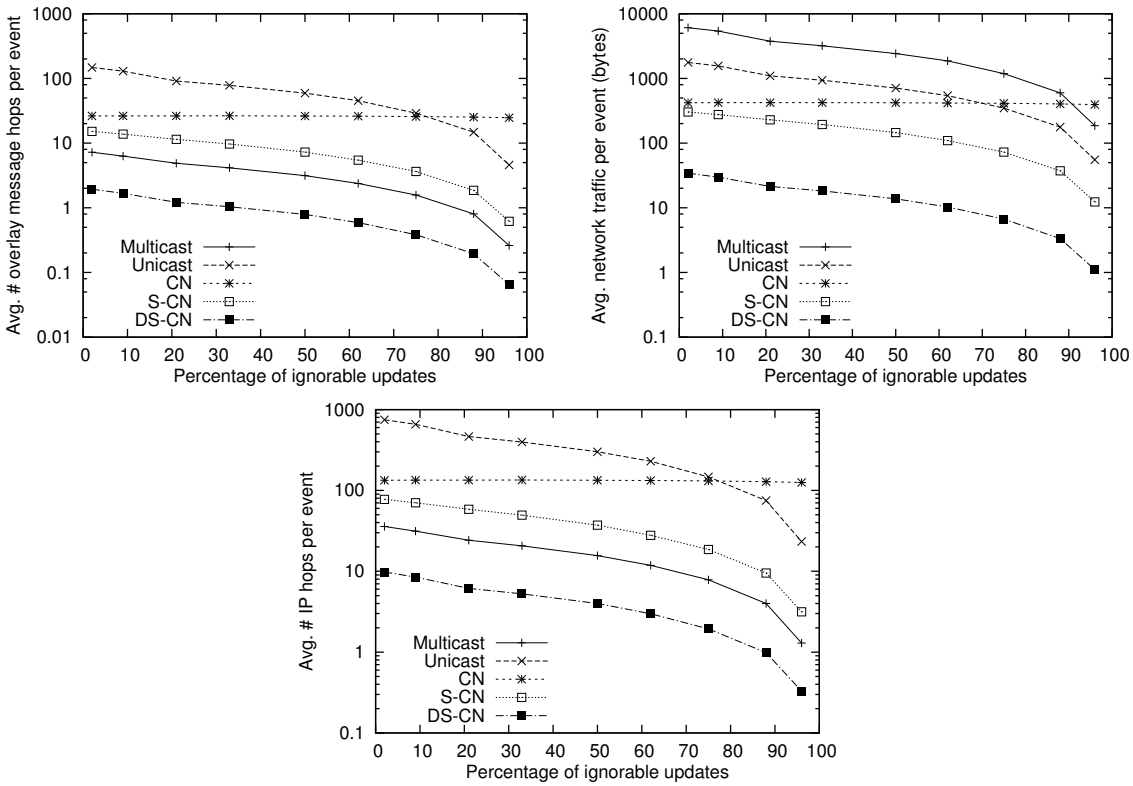


Figure 3.9: Performance; varying the percentage of ignorable updates.

(as discussed earlier) and would cause more traffic. We do not advocate CN$^+$ because it pushes a significant amount of complex application-specific routing logic into the network layer.

Varying percentage of ignorable updates

We next demonstrate the effect of increasing the percentage of ignorable updates I. To better control the parameter, we use a subscription distribution where the midpoint of the selection range is taken from a normal distribution $N(5000, 10000)$ and the length is taken from a normal distribution $N(50, 10)$. There are no spikes introduced. Figure 3.9 shows results for three network metrics. The left figure shows the average number of overlay hops per event, while the middle figure shows the average network traffic (in bytes) per event, for all approaches. The serverless CN approach is independent of I because we cannot truncate ignorable updates. As I is increased, the performance of all the approaches except CN improves and, for high values of I, CN becomes the worst approach.

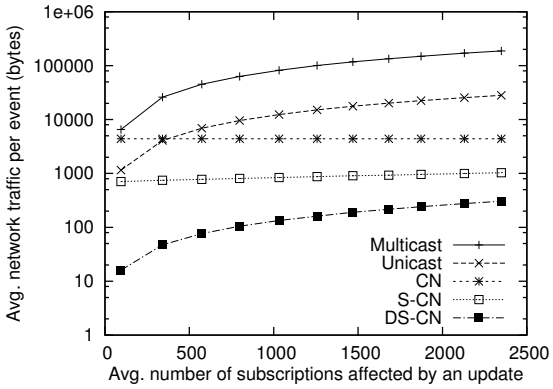


Figure 3.10: Traffic vs. # affected subs.

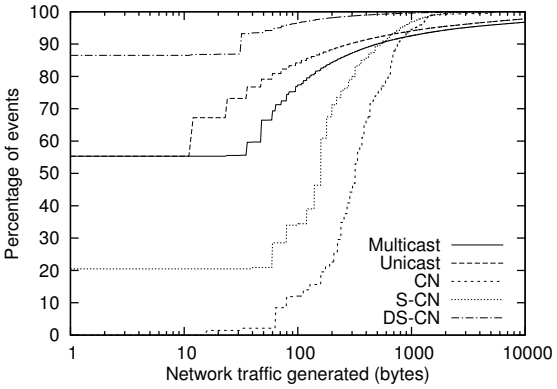


Figure 3.11: CDF of network traffic.

For high values of I, serverless approaches are less desirable as they are unable to detect and truncate ignorable events. However, DS-CN still performs well because in most cases where an update is ignorable, it can be detected at the zone owner containing the event point itself. Overall, the best performance is achieved by DS-CN. Multicast performs quite well in terms of number of overlay messages, with S-CN following up in third position. However, as the middle figure shows, multicast is the worst in terms of total network traffic (due to message size). The right figure shows the average number of IP-level hops per event, for each of the various approaches. The trends are similar to those of overlay hops. In further experiments, due to space reasons, we show only the average network traffic (in bytes) for various approaches; the other metrics were found to follow similar trends as described here, in all our experiments. In summary, the single server-based S-CN and the distributed DS-CN perform the best as the messages encode the affected subscriptions very compactly and messages are sent only to those CAN regions which could be affected by the event.

Varying number of affected subscriptions

We increase the average number of subscriptions affected by a non-ignorable update by controlling the percentage of spikes which affect a large number of subscriptions. All updates are non-ignorable. The number of subscriptions is $500,000$ and the database size is $50,000$ tuples. Figure 3.10 shows the performance of each of the approaches in terms of average network traffic (in bytes). From the figure, we see that unicast and multicast worsen in performance linearly with in-

Approach	Max. node stress
Unicast	$10,282$
Multicast	3
S-CN	23
DS-CN	25
CN	8

Table 3.4: Maximum node stress.

crease in average number of affected subscriptions. CN is independent of this parameter and hence shows a flat line. The network traffic generated by S-CN increases very slightly across the workloads as seen from the figure. This is due to larger average Mar as a result of increasing percentage of spikes. Finally, DS-CN shows an increasing trend; the reason is that since spikes have a large Mar, the diagonal traversal generates more traffic. In real workloads, such updates with large Mar are extremely rare.

Maximum node stress for various approaches

We compared the maximum node stress (MNS) for a workload with 21% ignorable updates, $500,000$ subscriptions, and $10,000$ database tuples. There were no spiked events. The highest MNS (across events) for unicast was found to be extremely high $(10,282)$ at the server. From the CDF of MNS over events (not shown), we found that although more than half the events generate no message, a fair percentage of events generate very high node stress at the server. This demonstrates the main disadvantage of unicast: lack of scalability for large-scale subscriptions. The highest MNS of S-CN and DS-CN were found to be just 23 and 25 respectively. In S-CN, 21% of events had 0 MNS, while in DS-CN, nearly 87% of events had 0 MNS. In both approaches, over 99% of events had a MNS less than 10. Multicast had a highest MNS of just 3 because the multicast trees are extremely skinny. Finally, CN had a highest MNS of 8. Although nearly every event has a non-zero MNS in CN, most events have a small MNS, for example, over 94% of events have a MNS of no more than 3.

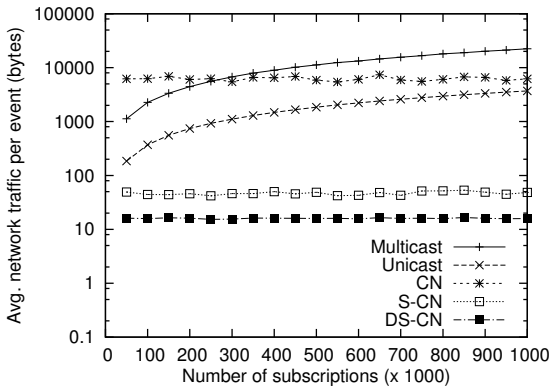


Figure 3.12: Result of real workload.

Distribution of network traffic

We plot the CDF (over events) of network traffic for all approaches, in Figure 3.11. There are $500,000$ subscriptions and $10,000$ database tuples. The percentage of ignorable updates is 21%. From the CDF, we see that S-CN, CN, and DS-CN perform well with average network traffic generated by an event less than $1kB$ for nearly all events. DS-CN is seen to be the most efficient as nearly 87% of events do not generate any traffic. S-CN also performs well but only the ignorable updates generate no traffic (other events need to be routed to the event point). However, nearly 97% of events generate traffic of less than $1kB$. The performance of CN is worse. Around 94% of events generate less than $1kB$ traffic, but more than half the events generate 320 bytes or more traffic. Unicast and multicast perform poorly overall as expected. However, these approaches perform well for events that affect few or no subscriptions: as a result, more than 77% of messages cause network traffic of fewer than 100 bytes. But, the performance degrades rather rapidly for the rest of the events, making these approaches very bad overall. Traffic of up to $680kB$ is seen for some events that affect a large number of subscriptions.

Experiments on a real workload

In order to evaluate our techniques on a real trace, we obtained information for 3053 stocks from Yahoo! Finance [Yah]. We gathered data for earnings per stock (`EPS`) for each of the stocks. In addition, computed the average recommendation over the past month (`Reco`) for each stock. `Reco`

varies from 1.0 (strong buy) to 5.0 (strong sell). We collected open and close price data over the course of 60 days, and used EPS to compute PER for each price. We thus obtained a trace of events, each being an update of PER with Reco constant. The trace had $338,415$ events. 11.7% of the events were non-ignorable events. Note that although this trace has only two events per day, real-time stock prices over the course of the day would follow a similar update trend, with several thousand updates generated every few seconds. This would need efficient database/network coordination to scale to large subscription sets.

We generated $500,000$ subscriptions; each subscription requests the minimum PER over a specified Reco range. This is a meaningful query because stocks with lower PER are intuitively better. Moreover, people may desire stocks rated at different ranges of Reco.

Figure 3.12 shows the average network traffic per event as we increase the number of subscriptions. We see that S-CN and DS-CN generate orders of magnitude lesser traffic than CN, unicast, and multicast. Both unicast and multicast do not scale well; their performances degrade linearly with increase in number of subscriptions. CN shows constant but bad performance. S-CN and DS-CN perform very well and are both independent of number of subscriptions. They generate less than 100 bytes of network traffic per event on the average, with maximum node stress never rising above 10.

3.6 Related Work

Publish/subscribe systems. Recently, research efforts have been focused on content-based publish subscribe systems which provide fine granularity and flexibility. These can be broadly characterized as systems where publishers publish events following a particular predefined schema, and subscribers express their interests as *profiles* [DRF04] which are predicates over the schema. A large number of such systems have been built in recent years, e.g., SIFT [YGM99] (for text documents), ONYX [DRF04] (for filtering and transformation of XML messages), and the wide-area event notification service [CRW01]. In all these systems, subscriptions are stateless filters defined over individual messages, so they cannot express queries of interest across different messages or

over the event history. Profiles are not powerful enough to accommodate stateful SQL-style subscription requirements. ONYX supports on-the-fly transformation of an XML message according to a subset of XQuery; but filtering and transformations are still limited to individual messages. A few continuous query systems also support rich query languages. Unfortunately, these do not address the problem of efficiently delivering updates over a network. ONYX has begun addressing this problem; however, the focus of ONYX on supporting transformation of XML messages is different from our goal of supporting more general stateful SQL subscriptions that cannot be processed on individual messages.

Database-side processing. Continuous query systems [LPT99, CDTW00] and stream processing systems [DEB03] can be regarded as a form of publish/subscribe where continuous queries over streams correspond to our subscriptions. These systems provide automatic notification whenever a continuous query result changes.

The idea of group processing has been identified and used in trigger processing and continuous query processing systems [HCH+99, CDTW00]. Work on scalable database trigger processing [HCH+99] focuses on exploiting common patterns in triggering conditions. Work on scalable continuous query processing (e.g., [CDTW00, MSHR02]) focuses on exploiting common patterns in continuous queries (like our subscription queries). In particular, predicate and query indexing techniques have been developed in [HCH+99, CDTW00, FJL+01] to speed up group processing. The upper hull computed for dissemination is similar to computing dynamic skyline in [PTFS05]. Their algorithm is based on regular R-tree, which cannot offer the same guaranteed performance as our A2B-tree.

Network dissemination. A server needs to deliver notifications to affected subscribers over a network. The problem of efficient message delivery has long been tackled in networking and distributed systems research. The traditional delivery mechanism is based on client polling. The next generation of delivery mechanism uses real push techniques based on group-based multicast protocols, e.g., IP multicast. Multicast provides a perfect interface for channel-based subscription services. IP multicast has also been exploited in building publish/subscribe systems that sup-

port more general filter-style subscriptions [OAA+00]. Because of slow adoption of IP multicast, there have been proposals for supporting application-level multicast using an overlay network (e.g., [CDKR02]). Oftentimes, they use distributed hash tables, which provide a convenient hash table abstraction over the participating overlay nodes. The problems with multicast were discussed in Section 2.3.1. Although we have included multicast as a comparison, the problem with most multicast techniques is that of number of groups. In order to implement publish/subscribe, we need a large number of groups (one for each possible subscriber set) and these techniques do not scale to large number of groups. Group reduction techniques [OAA+00] require the end subscriber to have the ability to filter out unwanted messages. The group size problem is seen in our experimental results with our implementation of multicast, where we avoided the need for large number of groups by encoding the set of subscriptions as part of the message.

An alternative dissemination interface is content-based networking [CW01, ASS+99, CF03a]. A content-based network can be used to implement a publish/subscribe system supporting filter subscriptions. A number of such systems have been developed (e.g., [TA04, TBF+03, GSAA04]). SemCast [PC05] proposes a number of techniques for efficient dissemination including the use of dynamic statistics. However, subscriptions in all these systems are limited to stateless filters. Nevertheless, they can still be used by our system as the messaging layer for delivering notifications once they are computed. We use message and subscription reformulation to enable traditional content-based networks to handle stateful queries such as range-aggregation.

3.7 Conclusions

In this chapter, we described and evaluated different solutions (on the spectrum of database/network interfaces) to efficiently handle range-aggregation and range-`DISTINCT` subscriptions in a large-scale publish/subscribe system. The solutions involved novel interfacing techniques between the database and the network, along with new algorithms and associated data structures. Experiments using a server and a large-scale simulated network showed the tradeoffs between the different approaches. Although unicast does not require state at subscriptions, the update traffic can be very

high. CN$^+$ can perform well, but introduces application- specific logic into the network and needs per-subscription state. S-CN and DS-CN perform well overall, with dramatic reduction in traffic at low server-side processing cost. We showed that using CN directly, to simply converting a stateful subscription to a stateless one, does not yield a scalable solution. Interestingly, we showed that it is possible to achieve orders-of-magnitude improvement (over other solutions) in terms of both network traffic and server-side processing costs.

Chapter 4

Supporting Value-Based Notification Conditions

4.1 Introduction

Today, we are faced with a huge increase in demand for personalized data. Millions of users request time-varying data like stock prices, auction bids, sports scores, etc. to be delivered to their cell phones, desktop clients, or email inboxes. The data needs are potentially different across subscribers, and thus translate to stateful subscriptions. One approach to supporting such subscriptions is to add post-processing logic and state maintenance at subscribers, but this approach is not scalable, as we will show in this chapter.

Consider, for example, an application that delivers stock price updates to a large number of subscriptions. While many subscribers may be interested in the same stock, each subscriber may have a unique data need in the form of a *notification condition*, which precisely specifies when the subscriber wishes to receive a price update. In this chapter, we focus on *value-based* notification conditions, where each subscriber is notified of the new price if and only if it differs from the price value last received by the subscriber by no less than a threshold (called *radius* and specified as part of the subscription definition). This problem setting corresponds closely to the well-known problem of *bounded approximate caching* [Ols03]. However, we have the stricter requirement that the subscriber (cache) should not be updated without a radius (bound) violation, and we also address the challenge of scaling to a large number of subscriptions.

Value-based notification conditions offer flexible, personalized control of when value updates are disseminated to subscribers. Besides stock price monitoring, value-based notification conditions are also useful in scenarios such as monitoring auctions and online sales prices, tracking sports scores and vote counts in elections, etc. Subscriptions with these notification conditions can also be used as building blocks for implementing wide-area approximate caching.

Challenges Support for value-based notification conditions in a publish/subscribe system involves the standard challenge of scalable subscription processing and notification dissemination, in the presence of millions of subscriptions. Naive solutions do not scale:

- Given an incoming event, if we let a server check all subscriptions in turn and notify each affected one with unicast, processing and dissemination costs can easily overwhelm the server.
- If we disseminate all events to subscribers and rely on post-processing at each subscriber to enforce notification conditions, there will be excessive network traffic, most of which is unnecessary.

To develop more efficient group processing and dissemination techniques, we are faced with several unique challenges raised by value-based notification conditions. First, each such subscription has a potentially unique per-subscription state that may need to be separately maintained—the last value that was sent to that particular subscription. This state can be different for different subscriptions, even for those with identical radius. Second, there is no simple subsumption relationship among subscriptions based on their radii. In other words, an update that needs to be sent to a subscription with a larger radius should not necessarily be sent to a subscription with a smaller radius (or vice versa). Even subscriptions with the same radius may receive completely different sequences of updates, depending upon when these subscriptions were created. We will explain these subtleties further using examples in Section 4.2.

Contributions We develop and evaluate a set of techniques for supporting scalable processing and dissemination for a large number of subscriptions with value-based notification conditions. Compared with less sophisticated alternatives, our techniques show a huge performance improvement in server processing, and an order of magnitude lower network cost for disseminating notifications.

As motivated in earlier chapters, we advocate a clean interface between the server and the network. All our techniques leverage standard or readily available network substrates such as IP unicast and content-driven networks (e.g., content-based networks [CW01], distributed hash tables

such as CAN [RFH+01], etc.) Accordingly, our solutions can be deployed easily and quickly, without worrying about pushing complex application processing logic inside the network.

We support notification conditions with arbitrary and different radii. Subscriptions can be added and removed dynamically, and we handle the case where subscriptions with identical radius may need to notified differently because their different creation times can lead to different per-subscription states. We adhere to a strict interpretation of notification conditions; i.e., a subscriber is notified if and only if the incoming event causes a radius violation.

We also propose two extensions, including 1) a more flexible form of notification condition specifying when a subscription must be notified, when it must not be notified, and when it may be optionally notified; 2) relative notification conditions, where notification is needed when the difference between the current and the last-notified values, measured in relative terms such as percent change, exceeds the prescribed threshold.

Outline The rest of this chapter is organized as follows. In Section 4.2, we describe our problem setting and semantics of value-based notification conditions precisely; we also warm up to our main discussion by presenting a series of less sophisticated solutions. Sections 4.3–4.6 present our solutions in a step-by-step manner. In Section 4.3, we tackle the first subproblem of handling subscriptions with different radii, assuming that no two subscriptions have the same radius. In fact, this solution works without the assumption, if we slightly relax the semantics of subscription creation to ensure that subscriptions with the same radius see an identical sequence of notifications. Without relaxing the subscription creation semantics, however, we need the solution to the second subproblem, described in Section 4.4, which handles subscriptions with same radius but different views of the data. Next, Section 4.5 describes how to combine the solutions to the subproblems in a single system, which provides full support for value-based notification conditions. Finally, Section 4.6 presents extensions of our techniques for flexible may-notify and must-notify conditions, and for relative notification conditions. In Section 4.7, we present a thorough experimental evaluation of our techniques. We discuss related work in Section 4.8 and conclude in Section 4.9.

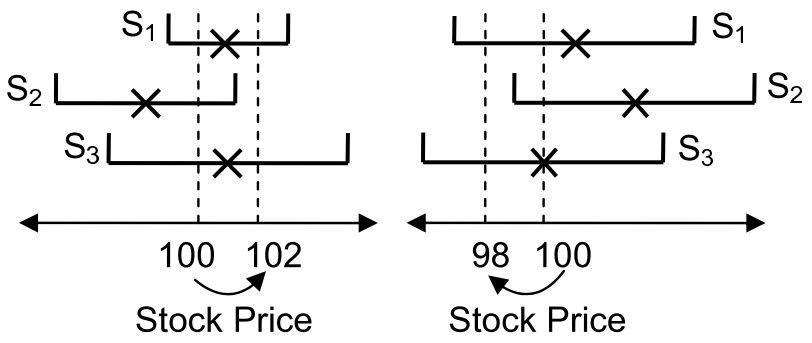


Figure 4.1: Example subscriptions.

4.2 Preliminaries

4.2.1 Semantics of Value-Based Notification

Consider a total of N subscriptions interested in tracking the value of the same data item over time. We denote the value of the data item at time t by $d(t)$. When a subscription is created, it is notified with the data value at the time of creation. Each subscription S_i specifies a *radius*, denoted by r_i. We notify the subscription with the current value of the data item if and only it has drifted by no less than r_i from the value last received by S_i.

Formally, at time t, the *center* of a subscription S_i, or its *last-notified value*, denoted $c_i(t)$, is the last value received by S_i at or prior to t. We call $(c_i(t) - r_i, c_i(t) + r_i)$ the *subscription interval* of S_i at t. The system maintains the invariant $d(t) \in (c_i(t) - r_i, c_i(t) + r_i)$ for any i and t (since the creation of S_i). If an update event causes $d(t)$ to fall outside S_i's subscription interval, we notify S_i, thereby *recentering* its subscription interval at $d(t)$.

The following two examples illustrate some of the intricacies that arise in supporting valued-based notification conditions.

Example 3 (Subscriptions with different radii)**.** *Suppose there are three subscriptions interested in tracking the price of a stock, with radii* 2, 4, *and* 6 *respectively. Figure 4.1 (left) plots these subscriptions as intervals* $(c_i(t) - r_i, c_i(t) + r_i)$*, with their centers indicated by crosses. The only obvious constraint on these intervals is that all of them must be stabbed by* $d(t)$*. In the figure, for*

instance, a stock price change from 100 *to* 102 *will affect the subscription with radius* 4 *but not the subscriptions with radius* 2 *or* 6*. Hence, there is no simple subsumption relationship implied by the lengths of subscription radii.*

Example 4 (Subscriptions with same radius)**.** *Consider three subscriptions with the same radius, say* 4*. The subscriptions were created when the stock price was* 101*,* 103*, and* 100 *respectively, as depicted in Figure 4.1 (right). If the current stock price is* 100 *and it changes to* 98*, we would notify* S_2 *but not* S_1 *or* S_3*. In fact, it is not difficult to devise a sequence of update events such that the sequences of notifications received by these three subscriptions are completely disjoint from each other. For example, each update event can always drop the value just below the largest (but above the second largest) left endpoint of the current subscription intervals; this sequence will cause the subscriptions to be notified in a round-robin fashion, one at a time.*

4.2.2 Notification Dissemination

For each incoming event, the publish/subscribe system identifies the subset of subscriptions that need to be notified, and uses a network substrate to deliver notification messages to the subscribers over a wide-area network. The details regarding our publish/subscribe model are present in Section 2.2.1. Briefly, we employ a network of brokers, each of which is responsible for a subset of the subscriptions. A broker forwards notifications to relevant subscriptions under its responsibility using whatever end-user delivery mechanism is suitable (e.g., IP, emails, instant messages). We primarily concern ourselves with the network substrate spanning all brokers, and ignore the "last-hop" delivery because it mainly depends on subscribers' needs and is orthogonal to the design of the rest of the system.

The most basic network substrate we can employ is the Internet, which supports IP unicast from a central subscription server to any individual broker. We also consider the general class of networks which we introduced in Section 2.2, called content-driven networks (CN). Recall that messages in CN do not specify any physical destination address; instead, they contain a list of attribute-value pairs. Destinations declare their interests with CN as boolean predicates over contents of individual messages. CN automatically routes messages based on their contents to destinations interested in

them.

While simple and easy to implement and deploy, CN does not directly support stateful subscriptions where processing requires information beyond individual messages. For subscriptions with value-based notification conditions, the missing information needed to determine whether a subscription is affected by an incoming event (i.e., whether the subscription needs to be notified because of the event) is the current center of the subscription. Despite this difficulty, we want to develop techniques for handling stateful subscriptions using CN for notification dissemination, in order to leverage CN's scalability and in-network predicate matching capabilities. The previous chapter considered other types of stateful subscriptions (e.g., range-min) without notification conditions; this chapter focuses on subscriptions with personalized value-based notification conditions.

Besides unicast and CN, another possibility is to push application state and processing logic into the network substrate, so that it supports stateful subscriptions directly (refer to CN^+ in Chapter 2). Systems such as *SMILE* [JS03] take this approach. We, on the other hand, consciously take the simpler approach of relying on standard network substrates that do not need to support application state, thereby preserving a clean, untangled interface between application and network. We believe this approach makes our system easier to deploy and maintain on a very large scale.

4.2.3 Alternative Approaches

Before describing our approach in detail, we warm up by presenting two alternatives that are more obvious and less sophisticated, followed by a discussion of some existing solutions [SRS02, SDR03].

B-tree(lr)+Unicast: Server-Based Processing with Unicast Dissemination In a subscription server, we can store all subscriptions for the same data item in a standard B-tree by indexing the left and right endpoints of each current subscription interval (hence the name *B-tree(lr)*). In other words, at the current time t, the B-tree stores the points $c_i(t) - r_i$ and $c_i(t) + r_i$ for each subscription S_i. Suppose an update event $d(t') \rightarrow d(t)$ arrives, where t' is the time of the last update and $d(t')$ denotes the value before the current update. We look up $d(t)$ in the B-tree, and

traverse leaf nodes towards $d(t')$. The subscriptions whose endpoints we encounter when traversing the leaf nodes are precisely those needing notification. To see why, note that if $d(t) > d(t')$, then these subscriptions are precisely those with $c_i(t') + r_i \leq d(t)$; if $d(t) < d(t')$, then these subscriptions are precisely those with $d(t) \leq c_i(t') - r_i$; in either case the new value has fallen out of the subscription interval.

Once identified, each affected subscription S_i is notified using unicast from the server. Since the notification recenters the subscription to $c_i(t) = d(t)$, to maintain the B-tree, we need to remove the old endpoints of S_i and insert the new ones $d(t) - r_i$ and $d(t) + r_i$. Recall that N denotes the total number of subscriptions. Let k denote the number of subscription affected by the incoming event. The cost of B-tree lookup and maintenance for the incoming event is $O(k \log_B N)$ I/Os, where B is the block size.

There are two problems with this approach. First, besides lookup, every incoming event requires deleting and reinserting each affected entry, which can have a high overhead if there are many affected subscriptions. Second, this approach enumerates the list of affected subscriptions and unicasts to each of them in turn. Because the server needs to send out these messages, it will experience heavy node stress and cause long notification delays for subscriptions that appear late in that list. The resulting network traffic will also be high. The server can batch all unicast messages destined to the same broker into one to save some overhead, but that one message still needs to enumerate all affected subscriptions hosted by the broker, so batching cannot avoid this problem fundamentally.

CN(lr): Serverless Processing and Dissemination by CN Although a value-based notification condition is stateful, we can make it stateless by instantiating the condition using the current subscription center. Specifically, each subscription S_i is defined by the predicate $(D \leq c_i(t) - r_i) \vee (c_i(t) + r_i \leq D)$, where D is the attribute in the update event message that stores the new value of the data item. This technique allows us to use a content-driven network to automatically deliver an update event to affected subscriptions, without using any server. We call this approach *CN(lr)* because it is based on CN and it instantiates each subscription as a predicate involving the left and right endpoints of the current subscription interval.

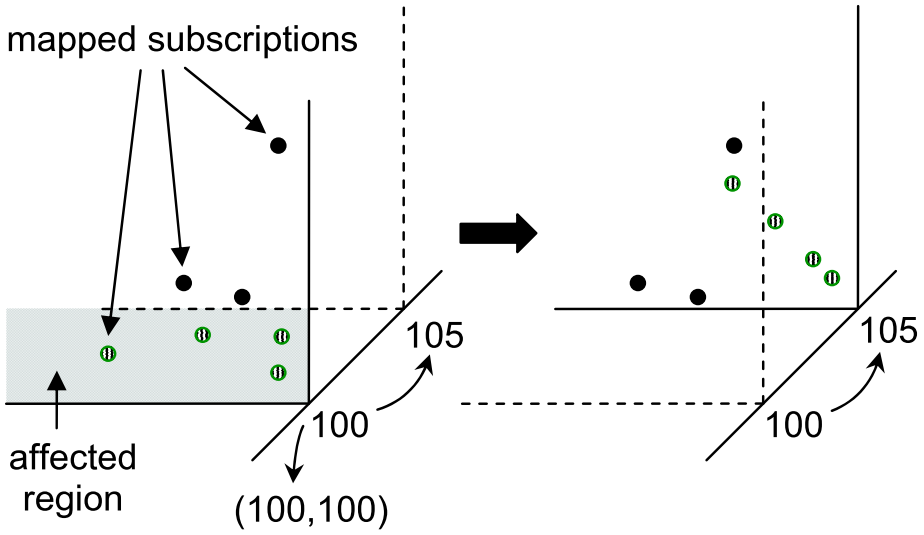


Figure 4.2: Value-based notification conditions over CAN.

As a concrete example, we show how to implement this approach using a specific CN: a two-dimensional CAN [RFH+01]. Each point (x, y) in this two-dimensional coordinate space represents an interval $[x, y]$. Each subscription S_i is mapped to the point $(c_i(t) - r_i, c_i(t) + r_i)$, as shown in Figure 4.2. Note that if a point p_1 is located to the northwest of another point p_2, then p_1's interval contains p_2's interval. Hence, all subscriptions in Figure 4.2 (left) are to the northwest of point $(100, 100)$, where 100 is the current data value. Suppose that an update event changes the value to 105. The affected subscriptions are those not containing the new value, shown in the shaded region.

CAN can be easily extended to support routing of messages to any designated hyperrectangle in its coordinate space, as was done in Chapter 2. Specifically, we partition the two-dimensional space into zones based on load-balancing criteria, and assign each zone to a CAN node. Each CAN node has knowledge of only its neighbors and can route messages only to them. We use CAN nodes as brokers, each of which is responsible for all subscriptions that fall within its assigned zone. When an update event $100 \rightarrow 105$ arrives, we first send it to $(100, 100)$ using standard CAN routing, and then forward it north and west through brokers until the shaded region is completely covered.

The problem here, however, is that each affected subscription S_i will be recentered and therefore must change it coordinate to $(d(t) - r_i, d(t) + r_i)$. This movement of subscription points is illustrated in Figure 4.2 (right). Unfortunately, this operation is expensive, because these subscrip-

tions will change brokers and potentially trigger load rebalancing.

Although there are other alternatives to CAN as the CN substrate (e.g., prefix hash trees and content-based networks), the efficiency problem caused by subscription recentering is universal. Even for CN substrates that do not require subscriptions to switch brokers (e.g., content-based network), changes of subscription intervals must be propagated through the network to update routing tables, which is also expensive. The fundamental problem is that while we have reformulated a stateful subscription to use a stateless definition, this definition depends on a dynamic property of the subscription (current center, in this case). Whenever this property changes, we may incur a significant overhead.

Existing Solutions Shah et al. [SRS02, SDR03] have investigated the problem of disseminating dynamic data in a network with the goal of meeting coherency constraints. The goal is to maximize *fidelity*, which is the percentage of time that the coherency constraints are met. Translated to the publish/subscribe setting, the only semantic requirement is that the value $c_i(t)$ at the subscriber S_i should lie within $(d(t) - r_i, d(t) + r_i)$. However, a subscription may receive an update even if the updated value does not fall outside the current interval. Furthermore, a subscription may even receive a value that never existed, as long as the resulting interval (recentered at this value) still contains the true value of the data item.

Their solution uses a customized dissemination tree with smaller radii serving larger ones. An event is disseminated starting at the root, using one of the following techniques:

- At the server, determine the largest radius affected ($r_{\max}$) and send the event to all S_i such that $r_i \leq r_{\max}$ [SRS02].
- *Distributed dissemination*, where a parent S_i sends an event to its child S_j if $|c_j(t) - c_i(t)| > r_j - r_i$ [SRS02]. Both this technique and the first one may result in unnecessary notifications which is forbidden by our stricter notification semantics.
- Enforce *dependent ordering* by sending *pseudo-values* [SDR03]. In this case, subscriptions may receive values that have never been published, which violates our publish/subscribe semantics.

Our precise notification semantics in Section 4.2.1 require that S_i should not receive any updates that do not fall outside the current subscription intervals. Furthermore, a subscription should only receive "true" values, i.e., values that have been produced by the publisher. We show that we can handle the precise semantics efficiently. We also discuss, in Section 4.6, how to relax the semantics to allow additional may-notify (loose) conditions in subscriptions. Another notable difference is that we use only standard network substrates, instead of using a customized dissemination tree that introduces complexity by adding application-specific logic inside the network,

4.3 Subscriptions with Different Radii

As the first step towards a complete solution, we assume in this section that all subscriptions with the same radius have identical centers. Under this assumption, by assigning subscriptions with the same radius to the same broker, we can effectively treat them as a single subscription, so all subscriptions now have unique radii.

This assumption is actually not as restrictive as it may appear at first glance. We can ensure that this assumption holds by enforcing the simple rule that a new subscription created at time t of the same radius r_i as some existing subscription S_i is sent an initial data value of $c_i(t)$ instead of $d(t)$. This alternative semantics of subscription creation may be acceptable for some applications because 1) $c_i(t)$ was a true value of the data item at some point in the past, and 2) the distance between the true value $d(t)$ and the value $c_i(t)$ now seen by the new subscription is guaranteed to be less than r_i. Nevertheless, for applications that require more precise semantics, we will tackle the general case of multiple subscriptions with the same radius having different current centers in Sections 4.4 and 4.5.

4.3.1 Scan(r)+CN(r): Towards Radii Indexing

The approaches introduced in Section 4.2 have a glaring problem with indexing subscriptions using properties that are dynamic (for both server processing and network dissemination). The solution that comes to mind is to index subscriptions by their radii, as the radii do not change. However, it

is unclear how to process an incoming event efficiently using such an index, because whether an event affects a subscription depends not only on its radius but also its current center, which is no longer indexed.

As a first step towards indexing by radii, we propose a simple technique called *Scan(r)*, which can produce not only a list of affected subscriptions (intended for unicast), but also a semantic description of affected subscriptions (intended for CN). Scan(r) maintains all subscription in a sorted order of increasing radii. Upon receiving an event, Scan(r) makes a linear scan and identify the subscriptions affected by the event. As it proceeds, Scan(r) can easily produce as its output a list of m *affected radius ranges* of the form $[\texttt{R\_LOW}, \texttt{R\_HIGH}]$, which indicates that all subscriptions with radius in this range is affected by the event. In Example 3, for instance, event $100 \rightarrow 102$ produces a single affected radius range $[4, 4]$. Scan(r) takes $O(N/B)$ I/Os, and does not need expensive updates to recenter subscriptions, as radii remain unchanged.

On the network side, we can "index" subscriptions by radii as well. Each subscription S_i is identified by its radius r_i, or more precisely, by predicate $\texttt{R\_LOW} \leq r_i \leq \texttt{R\_HIGH}$. We use a content-driven network to manage these subscriptions, and call this approach CN(r). Because radius is a static subscription property, CN(r) does not have CN(lr)'s problem of huge reorganization costs during subscription recentering. The list of m affected radius ranges, computed by Scan(r), works perfectly with CN(r). We inject each affected radius range as a message with attributes R_LOW and R_HIGH (in addition to the new data value) into CN(r), and CN(r) will automatically route it all subscriptions with predicates matching this message. Any incarnation of CN that supports range lookups (e.g., PHT or content-based network) can be plugged in.

This trick is an example of reformulation (see Chapter 2 for details), the idea of reformulating events as messages concisely describing subsets of affected subscriptions so that stateful subscriptions can be handled by network substrates that only directly support stateless subscriptions. In our case, Scan(r) uses the subscription state it maintains to reformulate an incoming event into messages that can be directly handled by CN(r). Note that for CN(r) to outperform unicast, we need m, the number of affected radius ranges, to be much less than k, the number of affected subscriptions. In our experiments (Section 4.7), we have found m to be at least an order of magnitude smaller than

k. The intuition behind this observation is that subscriptions with similar radii tend to be affected in batches. Even if two adjacent radii are separated by an update that cuts across them, there is a high likelihood that a future update will bring their centers very close together.

4.3.2 B^{A}-tree(r): Augmented B-Tree on Radii

We now present a data structure called *B^A-tree(r)* capable of computing affected radius ranges much more efficiently than Scan(r). Each incoming event $d(t') \rightarrow d(t)$ is processed by B^{A}-tree(r) as a *delta* $\Delta_i = d(t) - d(t')$. We start with a B-tree that indexes all subscriptions by radii. The tree topology is static in the absence of subscription insertions and deletions. We augment the B-tree with additional fields that allow us to track the subscriptions centers. To maintain this information efficiently, we update a subtree only when some, but not all, subscriptions in the subtree need to move. If all or no subscriptions in the subtree are affected, the entire subtree can be marked as updated simply by updating the root of the subtree. To support this nontrivial bookkeeping, we augment each index entry i with 8 following fields. An example B^{A}-tree(r) is shown in Figure 4.3.

- $r_{\min}[i]$ and $r_{\max}[i]$ track the smallest/largest subscription radius in the subtree rooted at i. A leaf node entry has $r_{\min}[i] = r_{\max}[i] = r$, where r is the radius of the subscription indexed by this entry.

- $L_{\min}[i]$ and $R_{\min}[i]$ track the smallest (in absolute value) negative/positive delta that will affect at least one subscription in the subtree rooted at i. Their values are accurate as of the last delta that affects at least one (but not all) subscription in the subtree rooted at i's parent.

- $L_{\max}[i]$ and $R_{\max}[i]$ track the smallest (in absolute value) negative/positive delta that will affect every subscription in the subtree rooted at i. Their values are accurate as of the last delta that affects at least one (but not all) subscription in the subtree rooted at i's parent. For a leaf node entry, $L_{\min}[i] = L_{\max}[i]$ and $R_{\min}[i] = R_{\max}[i]$. For all nodes, initially $L_{\min}[i] = R_{\min}[i] = r_{\min}[i]$ and $L_{\max}[i] = R_{\max}[i] = r_{\max}[i]$.

- $\gamma[i]$ is the *drift factor*, initially set to 0. It accumulates a sequence of consecutive deltas that

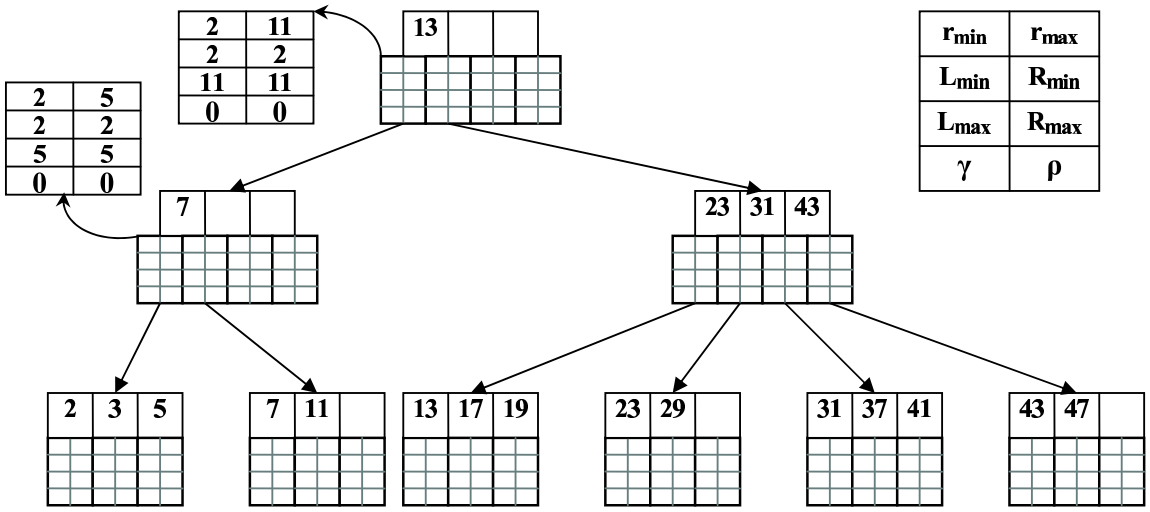


Figure 4.3: Example B^{A}-tree(r).

have reached this index entry but not yet been propagated down (because no subscription in the subtree is affected by any delta in the sequence).

- $\rho[i]$ is the *reset bit*. When set to 1, it indicates that all subscriptions in this subtree were affected by the last delta that affected at least one subscription in this subtree.

Let *anc*(i) be the set of ancestor index entries of i in the tree. There are two cases for how the reset bit is used to implicitly keep track of subscriptions in subtrees which are not directly updated.

- Case 1: For all $j \in$ *anc*(i), $\rho[j] = 0$. In this case, let $d = \sum_{k \in \textit{anc}(i)} \gamma[k]$. Then, the *current* values of $L_{\min}[i]$, $R_{\min}[i]$, $L_{\max}[i]$, and $R_{\max}[i]$ should be $L_{\min}[i] + d$, $R_{\min}[i] - d$, $L_{\max}[i] + d$, and $R_{\max}[i] - d$, respectively.

- Case 2: There exists $j \in$ *anc*(i) such that $\rho[j] = 1$ and $\rho[k] = 0$ for all $k \in$ *anc*(j). Here, let $d = \gamma[j] + \sum_{k \in \textit{anc}(j)} \gamma[k]$. Then, the *current* values of $L_{\min}[i]$, $R_{\min}[i]$, $L_{\max}[i]$, and $R_{\max}[i]$ are effectively $r_{\min}[i] + d$, $r_{\min}[i] - d$, $r_{\max}[i] + d$, and $r_{\max}[i] - d$, respectively.

Lookup (and Bookkeeping) On an incoming event Δ, we look up the B^{A}-tree(r) for the set of affected radius ranges by invoking LOOKUP$(\mathcal{T}, \Delta, 0)$, where $\mathcal{T}$ denotes the tree root. Algorithm 3

shows this procedure. Note that this "lookup" also updates the tree by performing necessary bookkeeping to track subscription movements; however, the structure of the tree remains unchanged.

During lookup, we consider three cases at each index entry of the node being visited. If the entire subtree is unaffected (Lines 7–10), we basically update the drift factor and do not recurse down. If the subtree is partially affected (Lines 11–13), we have to recurse down the subtree. If a previous lookup has set the reset bit for this entry, it implies that the child node has stale information. Hence, we invoke LOOKUP recursively with reset flag set to true. This flag indicates to the child index entries that they need to reset their information. If there is an accumulated drift factor, we need to add it to Δ before the recursive lookup. After the recursion returns, we can change the reset bit and drift factor to 0 for this index entry. Finally, if the subtree is completely affected (Lines 14–17), we set the reset bit for that index entry, change the drift factor to 0, and generate the affected radius range, and continue without recursing down. Resets propagated down by parents are processed in Lines 4–6. We incrementally compute the information to be returned to the parent in Lines 18–19.

It is not difficult to see that the I/O cost of lookup is $O(\min(m \log_B N, N/B))$, where m is the number of affected radius ranges it outputs. Lookup avoids visiting an entire subtree if the subtree is completely unaffected or entirely affected. Hence, to compute each affected radius range, lookup does not need to examine any nodes other than those on the paths from the root to the two endpoints of the range. Bookkeeping piggybacks on lookup and incurs no additional I/Os. With minor modifications, lookup can also produce the complete list of k affected subscriptions (for unicast) in $O(\min(m \log_B N + k/B, N/B))$ I/Os.

Producing Maximal Radius Ranges To ensure that m, the number of affected radius ranges, is as small as possible, we can modify Algorithm 3 to merge adjacent radius ranges with no unaffected subscriptions in between. Merging can be implemented in a streaming fashion because LOOKUP produces the ranges in order. The idea is to delay the output of affected ranges and instead keep a running range that can be extended until we encounter an unaffected leaf/subtree (or the end of processing); at this point, we can output the running range. This extension produces maximal radius ranges as output and incurs no additional cost.

Algorithm 3: Lookup algorithm for $\mathsf{B^A}$-tree(r).

```
1  LOOKUP(node n, float Δ, bool ρ) begin
2      ⟨L^ret_min, R^ret_min, L^ret_max, R^ret_max⟩ ← ⟨∞, ∞, 0, 0⟩;
3      foreach index entry i of n do
4          if ρ = 1 then
5              ⟨L_min[i], R_min[i], L_max[i], R_max[i]⟩ ← ⟨r_min[i], r_min[i], r_max[i], r_max[i]⟩;
6              ρ[i] ← 1; γ[i] ← 0;
7          if −L_min[i] < Δ < R_min[i] then
               // entire subtree unaffected
8              L_min[i] ← L_min[i] + Δ; R_min[i] ← R_min[i] − Δ;
9              L_max[i] ← L_max[i] + Δ; R_max[i] ← R_max[i] − Δ;
10             γ[i] ← γ[i] + Δ;
11         else if −L_max[i] < Δ < R_max[i] then
               // subtree partially affected
12             ⟨L_min[i], R_min[i], L_max[i], R_max[i]⟩ ← LOOKUP(child[i], γ[i] + Δ, ρ[i]);
13             ρ[i] ← 0; γ[i] ← 0;
14         else
               // entire subtree affected
15             ⟨L_min[i], R_min[i], L_max[i], R_max[i]⟩ ← ⟨r_min[i], r_min[i], r_max[i], r_max[i]⟩;
16             ρ[i] ← 1; γ[i] ← 0;
17             output ⟨r_min[i], r_max[i]⟩;
18         L^ret_min ← min(L^ret_min, L_min[i]); R^ret_min ← min(R^ret_min, R_min[i]);
19         L^ret_max ← max(L^ret_max, L_max[i]); R^ret_max ← max(R^ret_max, R_max[i]);
20     return ⟨L^ret_min, R^ret_min, L^ret_max, R^ret_max⟩;
21 end
```

Inserting and Deleting Subscriptions Subscriptions can be inserted and deleted using the standard B-tree procedures, with minor modifications to maintain the augmented fields. The I/O cost for these operations remains bounded by $O(\log_B N)$.

4.3.3 Notification Dissemination

Like Scan(r), B^{A}-tree(r) works with both unicast and CN(r) for notification dissemination. For unicast, B^{A}-tree(r) produces a list of k affected subscriptions; for CN(r), it produces a list of m maximal affected radius ranges, which are reformulated as messages that can be routed by a vanilla CN implementation that supports range searches. Since usually $m \ll N$, B^{A}-tree(r) should perform much better than Scan(r).

4.4 Subscriptions with Same Radius

We now come to the second subproblem as outlined in Section 4.1. Consider a set of N_r subscriptions with the same radius r. When a new subscription is created, it is centered at the current data value. Thus, subscriptions with the same radius may have different centers and receive different notification sequences. In this section, we present our solution, starting with two strawman solutions.

4.4.1 B-tree(c)+Unicast and CN(c): Strawman Solutions

B-tree(c)+Unicast While B-tree(lr)+unicast from Section 4.2.3 can be applied here, we can improve it by exploiting the fact that all subscriptions have the same radius for this subproblem. Instead of indexing two endpoints, we index just the current center of each subscription in a standard B-tree (hence the name B-tree(c)). On an increasing update event $d(t') \rightarrow d(t)$ where $d(t) > d(t')$, we can compute the list of affected subscriptions for unicast simply by looking up subscriptions centered within $(d(t') - r, d(t) - r]$ (the case for decreasing updates is symmetric). Then, we delete these subscriptions and reinsert them into the tree at $d(t)$, because they are now recentered at this

value. Deleting and reinserting them one at a time would result in $O(k_r \log_B N_r)$ I/Os, where k_r is the number of affected subscriptions. Instead, noting that affected subscriptions are to be deleted from a contiguous range of the B-tree and reinserted into another contiguous one, we can perform the deletions and insertions in batches, giving an overall I/O cost of $O(\log_B N_r + k_r/B)$ per event including both output computation and tree maintenance.

CN(c) An alternative, analogous to CN(lr) from Section 4.2.3, is to use a content-driven network to manage subscriptions identified by their centers (hence the name *CN(c)*). On each increasing update $d(t') \rightarrow d(t)$, we simply inject a message with attributes $D' = d(t') - r$ and $D = d(t) - r$ into the CN, which automatically route it to affected subscriptions, whose centers satisfy $D' < c_i(t) \leq D$ (the case for decreasing updates is analogous). This approach does not require a server.

The main problem of CN(c), similar to CN(lr) and indeed common to all approaches that "index" dynamic subscription properties, is that recentering of affected subscriptions causes expensive maintenance overhead. In the case of CN(c), moving subscription centers lead to constant relocation of subscription and/or updates to routing information within the network. In Section 4.3, we were able to avoid this problem by indexing static radii. However, for subscriptions with the same radius, what static property can we use? The remainder of this section presents a solution.

4.4.2 Static Circular Ordering

On an incoming event, some subscriptions may be affected and need to be notified and recentered. Surprisingly, it turns out that while the subscription centers can move around, the relative ordering of these centers, when mapped onto a circle (we call this the *circular ordering*), remains static. Moreover, a newly created subscription does not break the circular ordering of existing subscriptions, but simply adds a new entry at some position in the ordering (similarly for removal of a subscription).

To explain the static nature of circular ordering, we start by proving an interesting property of subscription centers. It is clear that since all centers lie within a distance of r from $d(t)$, the distance between any two centers must be less than $2r$. We proceed to prove a stronger result.

Lemma 1. *The distance between any two subscription centers at any given time is less than the radius r; that is, $|c_i(t) - c_j(t)| < r$ for any i and j.*

Proof. Consider two subscriptions S_i and S_j, inserted at times t_i and t_j respectively. If $t_i = t_j$, then $c_i(t) = c_j(t)$ for all t and we are done. Let $t_j > t_i$. At time t_j, we have $c_j(t_j) = d(t_j)$ and $|c_i(t_j) - d(t_j)| < r$. Hence, $|c_i(t_j) - c_j(t_j)| < r$.

Now, consider an incoming event at some later time $t_k > t_j$. If this event affects both S_i and S_j, we have $c_i(t_k) = c_j(t_k) = d(t_k)$, so $|c_i(t_k) - c_j(t_k)| = 0 < r$. If this event affects neither of the two subscriptions, their centers remain unchanged, and so is the distance between them. Finally, if the event affects just one of them, say S_i, we have $c_i(t_k) = d(t_k)$ and $|c_j(t_k) - d(t_k)| < r$. Hence, $|c_i(t_k) - c_j(t_k)| < r$. □

Theorem 3. *Suppose that at time t, all subscriptions are arranged in increasing order of their centers. No future event or subscription insertion/deletion can change the circular ordering of these subscriptions.*

Proof. Case I (effect of events): An event that affects no subscriptions does not change any centers and cannot break their ordering. Consider an increasing update event $d(t_1) \rightarrow d(t_2)$ that affects at least one subscription. Such an event affects a set of k_r subscriptions with centers in the range $(d(t_1) - r, d(t_2) - r]$. However, note that there exists no subscription with a center of $d(t_1) - r$ or less; therefore, this event affects subscriptions with the k_r smallest centers. The new center for these subscriptions is $d(t_2)$. This center is at a distance of no less than r to the right of the smallest subscription center, because the right endpoint of that subscription, denoted S_0, was violated. However, from Lemma 1 we know that all other subscriptions are centered at a distance of less than r to the right of S_0's center. Hence, all affected subscriptions, which were those with smallest centers, now effectively become the subscriptions with largest center, and the circular ordering is not violated. Similarly, on a decreasing event, a set of subscriptions with largest centers effectively become those with the smallest center, preserving the circular ordering.

Case II (effect of subscription insertion/deletion): A subscription insertion or deletion at time t does not modify the current center of any existing subscription. Therefore, existing subscriptions

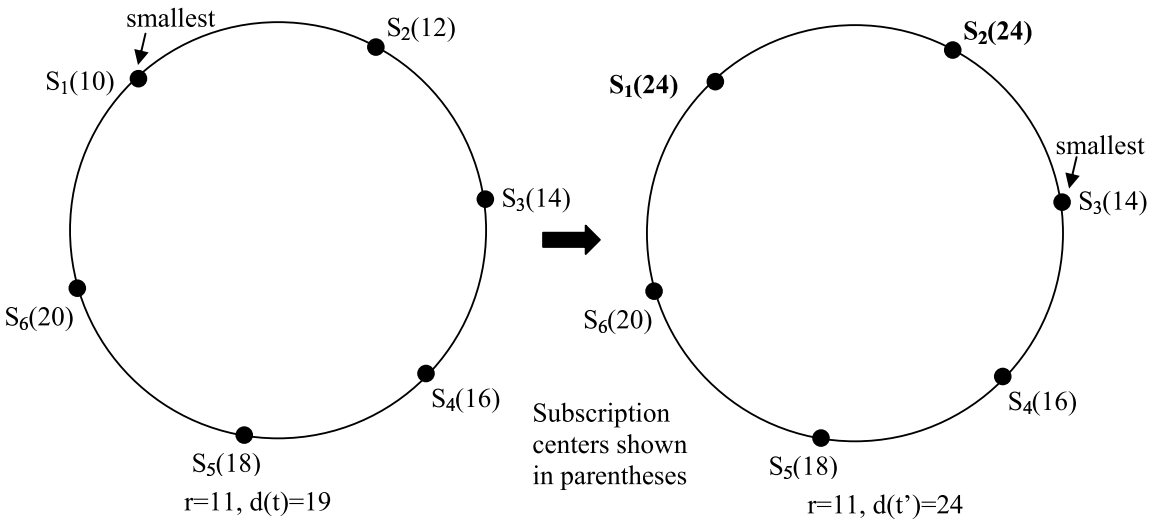


Figure 4.4: Static circular ordering of subscriptions.

retain their circular ordering. Insertion or deletion only changes the immediate neighbor for each of the two subscriptions whose centers are the closest to and on either side of point $d(t)$ in the circular ordering. □

To illustrate the static nature of circular ordering, Figure 4.4 shows six subscriptions of radius 11, arranged in increasing order of their centers, clockwise in the circle. If an update event of $19 \rightarrow 24$ arrives, the two subscriptions with the smallest centers (S_1 and S_2) are affected and are assigned the new center of 24. The circular ordering of subscriptions is unbroken, with S_3 now being the subscription with the smallest center (14).

4.4.3 B^W-tree(lid): Circular Augmented Weight-Balanced B-Tree

To index a set of subscriptions with the same radius, we assign an increasing, unique number to each subscription in the order of its current center. We call this number the *label* of the subscription. By Theorem 3, the circular ordering of labels does not change with data update events, even though centers can move. Also, as we have seen in the previous section, the set of affected subscriptions can always be expressed as a single range in the circular subscription space. Since the numeric label space is linear, a range in the circular space may be either a single interval (e.g., $[lid_1, lid_2)$)

or the union of two intervals (e.g., $(-\infty, lid_2) \cup [lid_1, +\infty)$) on numeric labels; in either case, two labels are needed to encode the range. Our goal is develop a data structure that is easy to maintain and can help us efficiently compute the two labels encoding the affected subscription range.

Our data structure is called *B^W-tree(lid)*, for *circular augmented weight-balanced B-tree on labels*. We first describe the basic data structure, and then discuss the modifications needed to maintain the labels in the face of subscription insertions and deletions. We start with a B-tree that indexes subscriptions by their labels. Every index entry i is augmented with the following:

- $c_{\min}[i]$ is the smallest subscription center in i's subtree.
- $c_{\max}[i]$ is the largest subscription center in i's subtree.
- $\rho[i]$ is a *reset bit* indicating that all subscriptions in i's subtree were affected by a single update and therefore have identical centers.

Over time, as the data item of interest is updated, any subscription can have the smallest center. We keep track of the subscription currently with the smallest center by remembering its label, called the *boundary label*. The index entries lying on the path from the root to the boundary label are called *boundary entries*. For an internal-node boundary entry, we maintain two sets of $c_{\min}$ and $c_{\max}$, for the two subranges separated by the boundary label. Figure 4.5 shows an example $\mathsf{B^W}$-tree(lid). Subscriptions are stored in leaves in the increasing order of their centers, starting with the boundary label and wrapping around at the end.

To ensure efficient lookup and maintenance, we use an idea similar to $\mathsf{B^A}$-tree(r) in Section 4.3: If subscriptions in a subtree are either all affected or all unaffected, we do not enter the subtree, but instead record the effect in the augmented fields of the subtree root. Recall from Section 4.4.2 that an increasing update event $d(t') \rightarrow d(t)$ affects all centers no greater than $d(t) - r$ (starting with the current boundary label) and relocates them to $d(t)$, which will become the largest centers located immediately before the new boundary label. To this end, we only need to take two paths down the $\mathsf{B^W}$-tree(lid): P_1, towards the current boundary label, and P_2, towards the smallest center greater than $d(t) - r$. In this process, for each node on P_1 (and P_2), we simply go through each index entry j located to the right (and left, respectively) of the entry leading to the next node on the path,

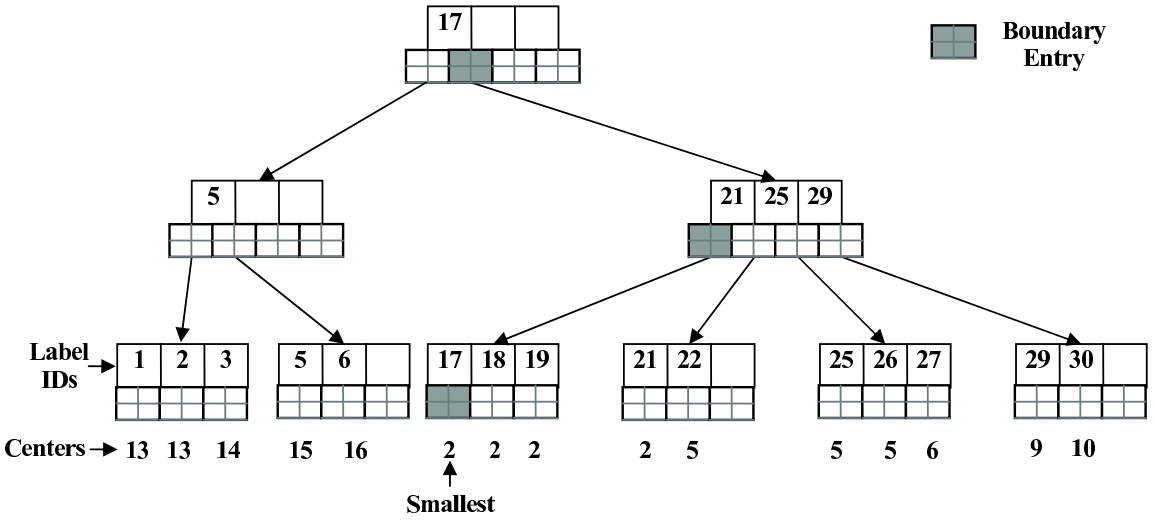


Figure 4.5: Example $\mathsf{B^W}$-tree(lid).

and set $\rho[j]$ and update $c_{\min}[j]$ and $c_{\max}[j]$ to $d(t)$. This method avoids updating subtrees that are completely affected. Maintenance of the boundary label and boundary entries also can be done during this process; we omit the details. Decreasing update events are handled similarly. Note that the entire process involves no changes to the tree structure.

Lookup on $\mathsf{B^W}$-tree(lid) returns two subscription labels encoding the range of affected subscriptions in the circular subscription space. The overall I/O cost, including any maintenance overhead, is $O(\log_B N_r)$ per event.

Inserting/Deleting Subscriptions A new subscription may be added between two adjacent existing subscriptions in the circular ordering. If there is a gap in the label space, a new label can directly be assigned to the new subscription. Otherwise, we need to relabel some existing subscriptions to make space for the new subscription.

A simple scheme such as relabeling all elements equally spaced over the label domain suffers from the problem of frequent relabeling, especially for skewed insertion patterns where new subscriptions are inserted repeatedly into the smallest gap. This *order maintenance* problem has been studied extensively in algorithms and database literature. We adopt the I/O-efficient relabeling scheme used in *W-BOX* [SHYY05], which was inspired by Dietz [Die82]. The relabeling scheme

is easily accommodated in B$^{\mathsf{W}}$-tree(lid) by using a weight-balanced B-tree [AV96] instead of a standard B-tree. The insertion cost (including relabeling) is $O(\log_B N_r)$ I/Os, amortized. Further improvements may be possible using alternative relabeling schemes such as [DS87], which bounds the number of relabelings per insertion by a constant.

4.4.4 Notification Dissemination

We use a content-driven network that manages subscriptions identified by their labels; we call this approach *CN(lid)*. For each incoming event, we use B$^{\mathsf{W}}$-tree(lid) to compute the two labels encoding the range of subscriptions to be notified, and inject into CN(lid) a single message containing these two labels as attributes. As before, any incarnation of CN supporting range queries is able to efficiently route this message to relevant subscriptions.

As discussed in the previous section, although events do not change subscriptions labels, we may need to occasionally relabel subscriptions in case we run out of labels when adding new subscriptions. For CN(lid), details and costs of relabeling depend on the particular CN used. Our implementation of CN(lid) in experiments, for example, uses a distributed B-tree. A relabeling operation, based on the W-BOX relabeling scheme we use (Section 4.4.3), can be concisely represented as a range of labels to be relabeled and the magnitude of shift, which is easily supported by our distributed B-tree. Relabeling does not alter routing paths because the circular ordering of subscriptions remains unchanged during relabeling. Only routing information covering relabeled ranges need to be updated. We investigate the overhead of relabeling in CN(lid) experimentally in Section 4.7.2, and show that it is low even at high subscription creation rates.

Alternatively, we can use unicast for notification dissemination in conjunction with B$^{\mathsf{W}}$-tree(lid). The lookup procedure of B$^{\mathsf{W}}$-tree(lid) can be modified to generate the list of affected subscriptions instead of a label range, which would take $O(\log_B N_r + k_r/B)$ I/Os, where k_r is the number of affected subscriptions.

4.5 Putting the Pieces Together

We now discuss how to support the most general case where subscriptions have arbitrary radii, and subscriptions with the same radius can have different centers. Consider the set of all subscriptions with the same radius. An event can affect (subscriptions with) this radius in three ways: 1) The radius can be completely affected, i.e., the event affects all subscriptions with this radius; 2) the radius can be completely unaffected; or 3) the radius can be partially affected.

The basic idea is to use a two-layer system. A *primary server* generates ranges of radii that are either completely or partially affected. A set of *secondary servers* are responsible for individual radii. Conceptually, these servers are organized as a CN(r) as in Section 4.3, where secondary servers are indexed by their designated radii. For each radius, its designated secondary server forms a CN(lid) with other brokers in the system, where each subscription is identified by its label, as in Section 4.4. We call this two-level network substrate *CN(r,lid)*. Although we have described CN(r,lid) conceptually as consisting of many content-driven networks, in practice they can be implemented by one network, with additional attributes encoding message types and additional predicates that distinguish them.

With CN(r), the primary server can send an update event to all affected secondary servers using messages containing the affected radius ranges. In order to compute these radius ranges, the primary server uses a two-level B-tree called the *B^2-tree(r,lid)*, composed of one upper-level B^A-tree(r) and a B^W-tree(lid) for each radius at the leaves. The processing cost for the primary server is $O(q \log_B N' + s \log_B N)$ per event, where q is the number of completely affected radius ranges, N' is the number of unique radii, s is the number of partially affected radii, and N is the total number of subscriptions. It may appear that the primary server needs to only maintain the B^A-tree(r). However, when a radius r is partially affected, the B^A-tree(r) would not be able to adjust L_{min}, L_{max}, R_{min}, and R_{max} fields for r because their values depend on the new smallest and largest centers among subscriptions with radius r, and this information is only available in the B^W-tree(lid) for r.

Each secondary server maintains a B^W-tree(lid) for each radius r it is responsible for. Upon

receiving from the primary server an event affecting r, the secondary server uses the $\mathsf{B^W\text{-}tree(lid)}$ for r to compute a label range identifying the set of affected subscriptions with radius r. This label range is injected into $\mathsf{CN(lid)}$ and routed to affected subscriptions.

4.6 Extensions

4.6.1 Generalizing Notification Semantics

We can extend our subscription language and semantics to allow for a range of radii $[r^-, r^+]$ specifying the acceptable range of deviation from the last-notified value. For a subscription S_i, the radius range $[r_i^-, r_i^+]$ replaces the rigid r_i threshold. We refer to r_i^- as the *may-notify* radius, and r_i^+ as the *must-notify* radius. S_i *must* be notified if $|c_i(t) - d(t)| \geq r_i^+$. In addition, S_i *may* be notified if $|c_i(t) - d(t)| \geq r_i^-$. Finally, S_i should definitely *not* be notified if $|c_i(t) - d(t)| < r_i^-$. The additional flexibility provided by this type of notification conditions opens up optimization possibilities.

For subscriptions with unique, different radii (as considered in Section 4.3), we seek to reduce the number of affected radius ranges. If two or more radius ranges are separated only by subscriptions that may be notified, we can concatenate these ranges into a single radius range. Disseminating a lesser number of larger radius ranges means that we pay the network overhead of reaching a radius range, less often. This reduces the overall network traffic. To this end, we change the $\mathsf{B^A\text{-}tree(r)}$ to sort all subscriptions by their may-notify radii, and set $r_{\min} = r_i^-$ and $r_{\max} = r_i^+$ for S_i's leaf node entry (as opposed to setting both to r_i in Section 4.3). Generation of radius ranges works the same way as in Algorithm 3. However, we do not output any isolated radius range consisting entirely of subscriptions that may, but do not have to, be notified.

For subscriptions with the same may-notify radii and the same must-notify radii (as considered in Section 4.4), we seek to notify all subscriptions that may be notified, as long as at least one subscription must be notified. The underlying intuition is that a notification will cause all notified subscriptions to have the same center, which means that they will behave identically from now

onwards and therefore can be effectively treated as a single subscription, thereby reducing the number of unique subscriptions in the system. We omit the details of modifying B^{W}-tree(lid) to implement this heuristic. Note that with this heuristic, Theorem 3 still holds because notifications are only made with respect to radius $r^- \leq r^+$, and notifications are performed only when some r^+ is violated.

4.6.2 Relative Notification Conditions

We now consider another extension of our subscription language to support *relative notification conditions*, where the radius is specified not in absolute terms but relative to the last-notified value. We discuss two types of such notifications.

The first type uses an *additive-relative radius*, which forms the subscription interval by subtracting from and adding to the subscription center. For instance, users may be interested in receiving a stock price update if the current price has deviated from the last received price by at least 10%. Formally, the subscription interval is given by $(c_i(t) - c_i(t) \cdot p_i, c_i(t) + c_i(t) \cdot p_i)$, where $p_i > 0$ is the *additive-relative radius* parameter. The subscription needs to be updated if $|d(t) - c_i(t)| \geq p_i \cdot c_i(t)$. In the case of subscriptions with unique, different p_i's, B^{A}-tree(r) from Section 4.3 works similarly, using p_i instead of r_i. The main change is that if an entire subtree rooted at index entry j is affected, we have to set $L_{\min}[j]$ and $R_{\min}[j]$ values to $p_{\min}[j] \cdot d(t)$ instead of $r_{\min}[j]$ (likewise for $L_{\max}[j]$ and $R_{\max}[j]$). In the case of subscriptions with the same p, however, we cannot directly use B^{W}-tree(lid). The reason is that Theorem 1 no longer holds due to the fact that the radius is dependent on the current center.[1] Nevertheless, we can still use the B^{A}-tree(r)-based solution if we relax the subscription creation semantics to ensure that subscriptions with the same radius have the same center, as discussed in the beginning of Section 4.3.

The second type of relative notification conditions uses a *multiplicative radius*, which forms the subscription interval by dividing and multiplying the subscription center. Formally, the subscription

[1] We can support the relative change specification on a B^{W}-tree(lid) by introducing a must-notify radius of $p_i^+ = p_i$ and a may-notify radius of $p_i^- = \frac{p_i}{1+p_i}$. However, the data structures and algorithms are more complicated, and outside the scope of this work.

interval is given by $(c_i(t)/f_i, c_i(t) \cdot f_i)$, where $f_i > 1$ is the *multiplicative radius* parameter, and we assume that the data value of interest is always positive. This type of relative notification conditions can be handled by transforming the values by taking their logarithms. In the logarithmic domain, the subscription interval becomes $(\log c_i(t) - \log f_i, \log c_i(t) + \log f_i)$, which is just a standard, non-relative value-based notification condition with radius $\log f_i$. Therefore, we can directly apply the techniques developed in the previous sections in the logarithmic domain.

4.7 Evaluation

4.7.1 Metrics, Workload, and Setup

For the *all-rad* case, where subscriptions can have arbitrary radii, and subscriptions with the same radius can have different views of the data, we we have implemented the following techniques for server-side processing:

- B-tree(lr): A simple B-tree constructed on left and right endpoints of current subscription intervals (Section 4.2.3).
- B^2-tree(r,lid): Our two-level data structure indexing subscriptions first by radius, and then by labels (Section 4.5).

For the *diff-rad* case (Section 4.3), where subscriptions with the same radius have identical view of the data, i.e., unique subscriptions all have different radii, we have implemented the following techniques (in addition to B-tree(lr), which is also applicable in this case):

- Scan(r): Storing subscriptions sorted by radii, and performing a linear scan to output radii or radius ranges (Section 4.3.1).
- B^A-tree(r): Our augmented B-tree constructed on subscription radii (Section 4.3.2).

Finally, for the *same-rad* case (Section 4.4), where subscriptions have the same radius but different views of the data, we have implemented the following techniques (in addition to B-tree(lr), which is also applicable in this case):

- B-tree(c): A simple B-tree constructed on subscription centers for subscriptions with the same radius (Section 4.4.1).

- B^W-tree(lid): Our augmented weight-balanced B-tree on the subscription labels (Section 4.4.3).

Although our data structures have been designed to be I/O-efficient, our current implementation uses main memory. The server processing time we measure in experiments mostly reflect CPU and memory access costs.

For the network substrate, we have implemented a simulator for large-scale networks. The first phase of network simulation generates application-level routing traces, which are further analyzed by a second phase to produce detailed costs. The second phase performs a link-level simulation using a topology produced by *INET* [CGJ+02], a generator of Internet-like network topologies. We consider both unicast and content-driven network (CN) for notification dissemination. The CN we have implemented is structured as a distributed B-tree with support for load balancing. The root of the B-tree is hosted by a server, and the nodes are hosted by brokers. We experiment with the following dissemination methods:

- Unicast from the central server. This method can be used in conjunction with B-tree(lr), B^2-tree(r,lid) (for all-rad), Scan(r) (for diff-rad), B^A-tree(r) (for diff-rad), B-tree(c) (for same-rad), and B^W-tree(lid) (for same-rad).

- CN(lr), serverless content-driven network where subscriptions are identified by the left and right endpoints of their intervals (Section 4.2.3). This method works in all three cases.

- CN(c), serverless content-driven network where subscriptions are identified by their current centers (Section 4.4.1). This method works for same-rad.

- CN(r), content-driven network where subscriptions are identified by their radii (Section 4.3.3). This method works in conjunction with Scan(r) and B^A-tree(r) for diff-rad.

- CN(lid), content-driven network where subscriptions are identified by their labels (Section 4.4.4). This method works in conjunction with B^W-tree(lid) for same-rad.

- CN(r,lid), two-level content-driven network (Section 4.5). This method works in conjunction with B^2-tree(r,lid) for all-rad.

Evaluation Metrics We use both server- and network-side metrics for evaluation. On the server side, we track processing time, which is measured as the duration between the time at which an update arrives at the server and the time at which the server completes generation of all outgoing messages for dissemination. On the network side, we track, for each event: 1) *Number of overlay message hops*, which measures the total number of messages sent between overlay nodes (for CN), or between the server and brokers hosting subscriptions (for unicast). 2) *Number of IP message hops*, which measures the number of hops over IP-level links. An overlay hop may involve traversing a number of IP-level links on its path. 3) *Network traffic*, which measures the total number of bytes transferred between overlay nodes (for CN) or between the server and brokers (for unicast). 4) *Maximum node stress*, which measures the number of messages originating from a node. The maximum node stress for an event is the highest node stress among all nodes while processing that event. Besides subscription processing and notification dissemination, these metrics also account for the overhead of recentering notified subscriptions, inserting/deleting subscriptions, and load balancing (for CN).

Workload We generate synthetic subscription radii using a truncated normal distribution to model skewed interests. The radii lie in the range $[1, 200\text{k}]$. In case of same-rad, we choose the initial centers of subscriptions with the same radius from a truncated normal distribution.

We experiment with both synthetic and real event data. Synthetic event data is derived using an order-1 autoregressive model, $\text{AR}_1(c, \phi, \sigma)$. The new value set by the i-th update is derived as: $U_i = c + \phi U_{i-1} + N(0, \sigma)$, where $N(0, \sigma)$ represents a normally distributed error with mean 0 and standard deviation σ. We present results for $\phi = 1$, which represents a drifting random walk. In addition, we experiment with historical stock prices collected from Yahoo! Finance [Yah]. Details of the real workload are described later.

Table 4.1 summarizes the main experimental parameters.

parameter	value
number of overlay nodes	1000
number of physical nodes	20000
number of events	100000
diff-rad	
number of subscriptions	10k–100k
subscription radii	$N(25000, 50000)$
distribution of events	$\text{AR}_1(400, 1, 1000)$
same-rad	
number of subscriptions	5k–50k
subscription radius	51000
distribution of events	$\text{AR}_1(500, 1, 800)$
all-rad	
subscriptions	See Sec. 4.7.2
distribution of events	$\text{AR}_1(500, 1, 800)$

Table 4.1: Summary of parameters.

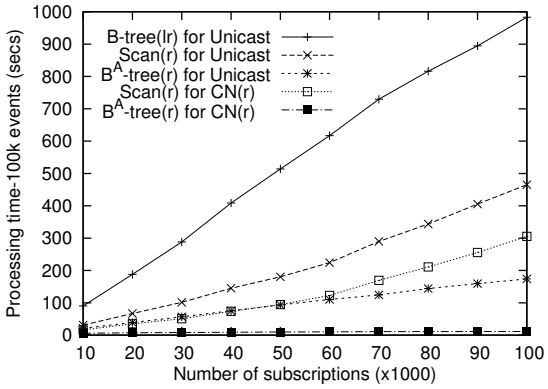


Figure 4.6: Processing time; increasing number of subscriptions (diff-rad).

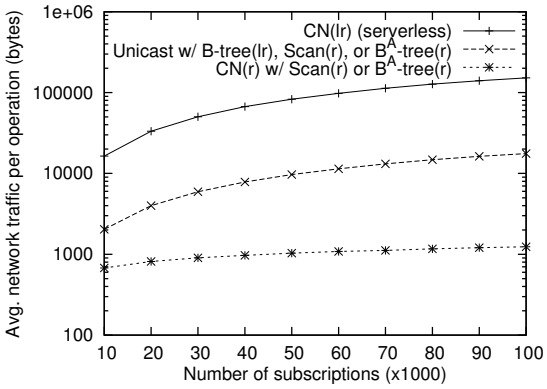


Figure 4.7: Network traffic; increasing number of subscriptions (diff-rad).

Experimental Setup We perform link-level simulation of an INET topology with 20000 nodes. Of these, 1000 nodes are chosen as brokers participating in the publish/subscribe system. All subscriptions are hosted by brokers. We do not model the last-hop cost for brokers to notify subscribers, because such costs are uniform across all approaches being compared, and heavily depend on the end-user delivery mechanisms such as emails and instant messages. Also, we do not model message hops from publishers to the server. Accordingly, to ensure fair comparison, we disregard the hop from the publisher to the network entry point for serverless approaches such as CN(lr) and CN(c).

In experiments, we have found results on IP-level costs (e.g., the number of IP message hops) to follow similar trends as those on node-level costs (e.g., the number of overlay message hops). Therefore, we only show results on node-level metrics.

4.7.2 Experiments and Results

Subscriptions with Different Radii (diff-rad)

We experiment with 100k synthetic events and from 10k to 100k subscriptions generated using the parameters shown in Table 4.1. Subscriptions with the same radius are constrained to have the same center in this subsection.

# subs.	20k	40k	60k	80k	100k
CN(r)	30	33	34	35	36
Unicast	499	982	1426	1850	2201

Table 4.2: Average number of outgoing messages from server (diff-rad).

Processing Time We increase the number of subscriptions and measure the server-side processing time for the various approaches that we implemented for diff-rad. From Figure 4.6, we see that B-tree(lr) really suffers because it indexes dynamic endpoints and incurs substantial overhead of recentering subscriptions. Although CN(r) is the ideal match for B^A-tree(r), B^A-tree(r) still outperforms B-tree(lr) and Scan(r) significantly even for unicast.

Other factors being equal, server-side processing techniques intended for unicast are generally slower than techniques intended for a content-driven network, because the latter techniques only need to generate a concise description of affected subscriptions instead of a long list. Table 4.2 shows the average number of outgoing messages from the server, for unicast and CN(r). We see that the semantic descriptions generated for CN(r) (radius ranges) are much more concise than lists of affected subscriptions, even after 100k events.

Back to Figure 4.6, we see that to produce radius ranges for CN(r), B^A-tree(r) performs much better than the naive Scan(r). B^A-tree(r) takes negligible time to process even 100k subscriptions.

Network Traffic We next compare the average network traffic generated per event in bytes for different notification dissemination methods, as we increase the total number of subscriptions. Figure 4.7 shows the results. Note that the y-axis uses a logarithmic scale. There is an order of magnitude reduction in network traffic when using CN(r), as compared with unicast. CN(lr) does even worse than unicast because of the overhead of recentering subscriptions as well as the added cost of load balancing necessitated by movements of subscriptions. Results on the number of overlay message hops reveal a similar trend, and are omitted.

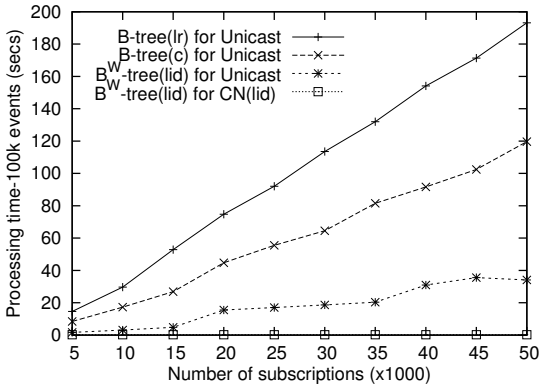


Figure 4.8: Processing time; increasing number of subscriptions (same-rad).

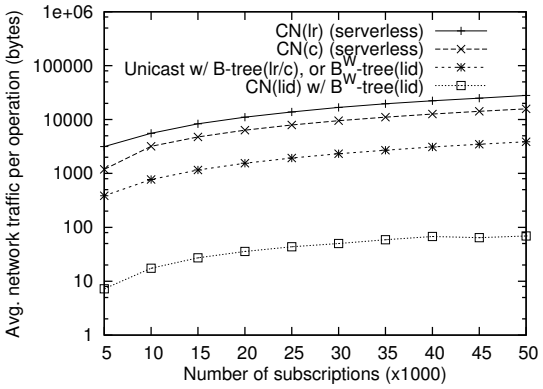


Figure 4.9: Network traffic; increasing number of subscriptions (same-rad).

Maximum Node Stress The maximum node stresses over all events (for 10000 subscriptions) are 3969, 1984, and 282 for CN(lr), unicast, and CN(r), respectively. CN(lr) is the worst because of the high reorganization cost caused by subscription recentering. Unicast bottlenecks the server with a large number of outgoing messages. CN(r) is the lowest because of the smaller number of radius ranges and the static content-driven network organization. For CN(r), 93% of events have maximum node stress less than 100.

To summarize, the combination of B^A-tree(r)+CN(r) offers the best solution to diff-rad. The serverless approach CN(lr) is too expensive in terms of network costs, while the other approaches are inferior to B^A-tree(r)+CN(r) by all performance metrics we have experimented with.

Subscriptions with Same Radius (same-rad)

We next experiment with a large number of subscriptions having the same radius; experimental parameters are shown in Table 4.1. We interleave arrivals of new subscriptions with events, so subscriptions can have different centers. During the bootstrap phase, we insert subscriptions that have initial centers normally distributed around the first event.

Processing Time We vary the total number of subscriptions from 5k to 50k, and measure processing time for 100000 events. The results are shown in Figure 4.8. We see that B^W-tree(lid) is very efficient at generating label ranges, taking negligible time regardless of the number of sub-

# subs.	10k	20k	30k	40k	50k
CN(lid)	0.04	0.04	0.04	0.04	0.04
Unicast	97	193	290	386	483

Table 4.3: Average number of outgoing messages from server (same-rad).

scriptions. B^W-tree(lid) also does very well at generating subscription lists for unicast, much better than B-tree(lr) and B-tree(c), which suffer because they index dynamic properties of subscriptions.

Table 4.3 shows the average number of outgoing messages generated by the server, for unicast and CN(lid). We see that the semantic descriptions generated for CN(lid) (label ranges) are extremely concise compared to lists of affected subscriptions. This factor also contributes to the lower server processing cost of B^W-tree(lid)+CN(lid).

Network Traffic In Figure 4.9, we show the average network traffic per event while increasing the total number of subscriptions for same-rad. Again, the y-axis uses logarithmic scale. Dissemination using CN(lr) is quite inefficient due to subscription movement and load balancing. CN(c) is slightly better as it indexes only the centers. Unicast, despite its simplicity, turns out to be better than both these alternatives. However, CN(lid) does orders of magnitude better, taking less than 100 bytes of traffic per event on average. The reason is that CN(lid) only needs to disseminate at most only two label ranges per event. Again, results on the number of overlay message hops reveal a similar trend, and are omitted.

Maximum Node Stress The maximum node stress over all events (for 10000 subscriptions) are 18051, 9026, 9025, and 2 for CN(lr), CN(r), unicast, and CN(lid), respectively. CN(lr) and CN(c) perform poorly due to reinsertion of all affected subscriptions. Unicast requires the server to send out a large number of messages. CN(lid) has the lowest node stress because it sends very concise semantic descriptions of affected subscriptions.

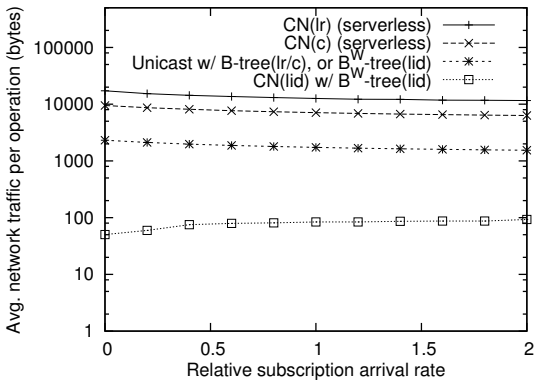


Figure 4.10: Increasing relative subscription arrival rate (same-rad).

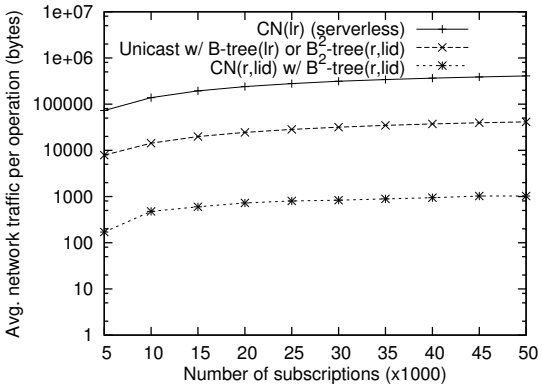


Figure 4.11: Average network traffic per operation (all-rad).

Performance of Insertion When subscriptions come and go over time, BW-tree(lid)+CN(lid) may need to perform subscription relabeling. Here, we test how relabeling costs degrade the performance of this combination. We allow subscriptions to be continuously inserted during the experiment, and vary the *relative subscription arrival rate (RSAR)*. RSAR of 2 means that two new subscriptions are inserted for every event arrival. We see from Figure 4.10 that the average network traffic per operation (subscription insertion or event dissemination) increases for CN(lid) with increasing RSAR. For the other approaches, subscription insertion is less expensive than event dissemination, which explains why the average cost of an operation decreases slightly with increasing RSAR for these approaches. Regardless, CN(lid) is orders of magnitude better than them, even for an unusually high RSAR of 2. In practice, we would expect the event arrival rate to be higher than the subscription insertion rate.

To summarize, the combination of BW-tree(lid)+CN(lid) offers the best solution to same-rad. Although serverless approaches incur no server-side processing cost, their high network costs make them infeasible. Other approaches are outperformed by BW-tree(lid)+CN(lid) in terms of both server- and network-side performance metrics.

Subscriptions with All Radii (all-rad)

Finally, we experiment with a large number of subscriptions with arbitrary radii and full personalization of event views. In this set of experiments, we generate subscriptions on stock prices as

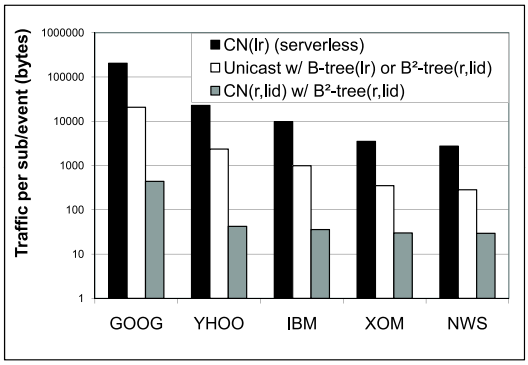


Figure 4.12: Average network traffic per operation, real workload (all-rad).

follows. To generate s subscriptions, we first have a baseline uniform random distribution of $0.1s$ subscriptions over radii (defined at cent boundaries) ranging from \$0 to \$100. To model the observation that subscriptions with radii at whole dollar amounts may be more popular (because they are simpler and more natural for users to specify), we overlay the remaining subscriptions as a truncated normal distribution $N(20, 20)$ constrained at dollar boundaries over the baseline set of subscriptions. We show performance only in terms of network traffic; the server processing cost of using B^2-tree(r,lid) for content-driven dissemination was found to be at least an order of magnitude lower than using B-tree(lr) for unicast.

Network Traffic We use the event distribution shown in Table 4.1. We first vary the total number of subscriptions from 5k to 50k, and plot the average network traffic per operation (new subscription or event) in Figure 4.11. The y-axis uses a logarithmic scale. We see that CN(r,lid) performs an order of magnitude better than unicast. As expected, the serverless CN(lr) does not do well because of subscription movements, performing even worse than unicast.

Results of Real Workload We use a real event workload in this experiment. Traces of historical daily stock prices were obtained from Yahoo! Finance [Yah], for 5 leading stocks from various industries. Each trace has between 600 and 11000 events. We replay the stock price variations in these traces multiple times, to get a dataset with 50k events per stock. We randomly interleave new subscriptions with incoming events.

Figure 4.12 shows the performance in terms of average network traffic per operation for this real workload. The y-axis uses a logarithmic scale. Again, the serverless CN(lr) does worse than the other approaches. Unicast does better, but our techniques show an order of magnitude reduction in traffic while adhering to the strict semantics of personalized notifications.

4.8 Related Work

Dynamic Data Dissemination As discussed in Section 4.2.3, Shah et al. [SRS02, SDR03] address the problem of disseminating dynamic data over a network of repositories. However, they target weaker subscription semantics and build a customized dissemination structure. We support stricter notification semantics efficiently, and leverage standard dissemination substrates.

Bounded Approximate Caching Bounded approximate cache maintenance [Ols03] has been studied extensively in database literature. Olston et al. [OW02] use bounded approximate caching to maintain caches on a best-effort basis, which does not adhere to precise subscription semantics. In [OLW01], the focus is on how to set these bounds adaptively based on query workloads. However, in our setting, notification conditions are specified by the subscriber and cannot be changed by the system. Also, related work in this area does not consider scalable indexing or update dissemination for bounded approximate caches.

Continuous Query Systems Continuous query systems [LPT99, CDTW00, TGNO92] can be regarded as a form of publish/subscribe system where continuous queries over streams correspond to our subscriptions. These systems provide automatic notification whenever a continuous query result changes. *OpenCQ* [LPT99] supports notification conditions that refer to current and previous database states, and *NiagaraCQ* [CDTW00] supports timer-based notification conditions. In other words, *NiagaraCQ* allows control over the staleness of subscriptions, but not over their accuracy, in terms of user-defined metrics. All these systems use simpler processing techniques for notification conditions and do not optimize notification dissemination. On the other hand, we propose efficient processing techniques and address dissemination issues as well.

Other Related Work A number of publish/subscribe systems built by the database community have made the subscription language more powerful [NACP01, DRF04]. Many of them have added language support for notification conditions. For instance, *Xyleme* [NACP01] supports *monitoring queries*, which are analogous to notification conditions. *SMILE* [JS03] supports SQL queries over the event history. However, none of them address the efficiency issues in processing notification conditions. Our techniques can be employed by such systems to support notification conditions in a scalable manner.

4.9 Conclusions

We address the problem of adding scalable support for subscriptions with personalized value-based notification conditions in a large-scale wide-area publish/subscribe system. Our first step was to efficiently process and disseminate events for subscriptions with varying radii, where subscriptions with the same radius share the same view of the data. We next showed how to efficiently support subscriptions with the same radius but different views of the data. Finally, we showed how to put the two pieces together to build a publish/subscribe infrastructure capable of handling the precise notification semantics in a scalable manner. We also discussed a number of extensions such as support for may-notify and must-notify conditions, and relative notification conditions.

We maintain a clean interface between the server and the network, and leverage established dissemination components. This design allows us to quickly build and deploy a robust wide-area publish/subscribe system. Our techniques were shown to be much more efficient than less sophisticated solutions, with an order of magnitude reduction in network traffic and server processing cost.

Chapter 5

Supporting Select-Join Subscriptions

5.1 Introduction

With an increasing number of data sources, there is a pressing demand for efficient support of complex subscriptions that correlate data across multiple sources. These subscriptions are stateful—given an incoming event, the system needs information beyond the content of this event itself in order to determine whether and how it affects these subscriptions. The relational join operator provides a convenient way to correlate data across sources, and *select-join* subscriptions are a common class of stateful subscriptions, as the following example illustrates.

Example 5. *Consider a publish/subscribe system for financial data. One source of data is basic stock information, conceptually represented by a table* $\texttt{Stocks}(\texttt{Symbol}, \texttt{PER}, \ldots)$. *In particular, the* PER *attribute records the price-to-earning ratio, a popular measure of stock quality. Another source of data is analyst reports, represented by* $\texttt{Reviews}(\texttt{Symbol}, \texttt{Rating}, \ldots)$. *In general, a stock may receive multiple ratings from different analysts.*

A subscriber may be interested in cases where a stock's rating and its PER *respectively belong to two prescribed ranges—for example, good ratings (no less than* 6 *on a scale of* 1 *to* 10*) for stocks with relatively high* PER *(between* 45 *and* 70*). This subscription, which we denote by* X_1*, can be expressed as a* select-join *query:*

$$\sigma_{\texttt{PER} \in [45,70]} \texttt{Stocks} \bowtie_{\texttt{Symbol}} \sigma_{\texttt{Rating} \in [6,10]} \texttt{Reviews}.$$

This subscription is stateful. Suppose an incoming event carries a PER *of* 60 *for a new stock. This event may or may not require the subscriber to be notified, depending on whether the stock has any rating at or above* 6*. In order to determine whether notification is needed, and if yes, to compute the new join result tuples for the notification, we must refer to the contents of* Reviews*, which are not part of the event itself.*

Challenges Continuing with Example 5, we further illustrate the challenges (and opportunities) that arise in supporting select-join subscriptions in a wide-area publish/subscribe system. Suppose the current contents of `Stocks` and `Reviews` are as follows. For simplicity, let us first assume that each incoming `Stocks` event is an insertion into `Stocks` (e.g., if we track historical information in `Stocks`).

`Stocks`

	Symbol	PER	...
s_1	GOOG	51.7	...
s_2	YHOO	51.2	...
s_3	AMZN	92.8	...

`Reviews`

	Symbol	Rating	...
r_1	GOOG	5.5	...
r_2	GOOG	6.0	...
r_3	GOOG	7.1	...
	...	...	...
r_{20}	GOOG	9.5	...
r_{21}	YHOO	7.5	...
r_{22}	AMZN	7.2	...
r_{23}	AMZN	7.8	...

Given a `Stocks` event, a naive approach is to compute, for each subscription, any change to the result of select-join subscription query, in the same manner as *incremental view maintenance* [GM99a]. If this change is not empty, we say that the subscription is *affected* by this event, and we send the change to the subscriber in a notification message. For subscription X_1 in Example 5, computing the change requires joining the new `Stocks` tuple with `Reviews`. For example, on an event inserting a new GOOG tuple $s_4 = \langle \text{GOOG}, 52.1, \ldots \rangle$, we would send X_1 a message containing $s_4r_2, s_4r_3, \ldots, s_4r_{20}$, which is the subset of $\{s_4\} \bowtie$ `Reviews` satisfying the selection conditions of X_1.

The first obvious problem with this approach is that of *large server output size*. The server is burdened by having to enumerate potentially many output tuples. This problem slows down processing and increases server load, and in turn reduces event-processing throughput and increases notification latency. Upon closer examination, this problem is caused by three sources of redun-

dancy:

- ***Result representation redundancy***. Within each notification message, the newly inserted tuple is repeated many times, once for each joining tuple. For example, the notification for X_1 when s_4 is inserted repeats s_4's content 19 times.

- ***Current-content redundancy***. This redundancy arises when some information in the notification message can be readily inferred from the current subscription content. In many application scenarios it is reasonable to assume that subscription clients in a publish/subscribe system maintain the current subscription content. In this case, information that can be inferred from this content does not need to be included in the notification message. The naive approach does not take advantage of this option. For example, because of s_1, X_1's subscription content should already contain $r_2, \ldots, r_{20}$ prior to the insertion of s_4; there is no need to send these joining `Reviews` tuples again.

- ***Inter-event redundancy***: This redundancy arises when some information in the notification message cannot be recovered from the current subscription content, but nevertheless has been previously transmitted to the subscriber due to an earlier event. For example, suppose that a sequence of new `Stocks` events for GOOG drop its `PER` to lower than 45, and the older GOOG tuples, s_1 and s_4, have been expired (i.e., deleted) from `Stocks` (e.g., if we maintain only a recent window of history in `Stocks`). At this point, the joining `Reviews` tuples $r_2, \ldots, r_{20}$ are no longer in X_1's content. However, a future event may raise GOOG's `PER` to above 45 again, and bring back these joining `Reviews` tuples. Ideally, we would like to avoid resending these tuples to X_1 as much as possible.

Another serious problem of the naive approach is the *lack of sharing of dissemination costs* across subscribers. A large-scale publish/subscribe system may have millions of subscribers, and many of them may be interested in similar join results. For example, another subscription X_2 may be interested in a `PER` range of $[50, 80]$ and a rating range of $[7, 10]$, which are different from but overlap with those of X_1. For insertion of s_4, changes to X_1 and X_2 are nearly identical—both contain $s_4r_3, \ldots, s_4r_{20}$, and the only difference is that X_2 should get s_4r_2. When notifying

multiple subscriptions, we wish to avoid ***inter-subscription redundancy***, which increases not only the overall communication cost, but also server stress and notification latency.

There is even more opportunity for improving the overall efficiency by identifying and avoiding ***re-dissemination redundancy***, which is a generalization of current-content, inter-event, and inter-subscription redundancies. For example, suppose that a third subscription, X_3, is interested in the PER range of $[80, 100]$ and the same rating range as X_1. Thus, r_{22} and r_{23} (reviews for AMZN) are in X_3. Later on, if AMZN's PER becomes 55, X_1 should receive r_{22} and r_{23}. Suppose that X_1 and X_3 share some portion of their network dissemination paths; it would be nice if we could avoid resending the same contents through the shared path. This optimization enables cost sharing across both events and subscriptions.

The solution to these problems is challenging. 1) Compressing individual notification messages can avoid result representation redundancy, but is ineffective at removing other redundancies. 2) We can remove current-content redundancy at the server by checking the content of each outgoing notification against the current subscription content. However, the overhead is high, and it is difficult to scale to a large number of unique subscriptions. 3) Inter-subscription redundancy can be overcome by a network of brokers, a popular approach in wide-area publish/subscribe systems. Recall from Chapter 2 that brokers are each responsible for a subset of the subscriptions, and they forward events to each other based on downstream subscription interests. Each event traverses down through a tree of brokers spanning all affected subscribers. However, traditional publish/subscribe systems do not directly support stateful subscriptions such as joins, and their stateless nature makes it difficult to implement state- and history-based optimizations, such as avoiding current-content and inter-event redundancies. 4) No existing solutions or simple extensions (discussed in Section 5.2) are able to avoid all types of redundancies. Furthermore, these solutions do not mesh well with each other, so it is unclear how to combine them to simultaneously avoid redundancies of different types, or the general re-dissemination redundancy.

Design for Practicality Besides the challenges above, there are important practical considerations as outlined in Chapter 1. We do not wish to over-complicate the design of broker networks, e.g., by augmenting brokers with hard application state or application-specific processing logic, or

general database-like processing capabilities. While complex, stateful networks may be appropriate in certain scenarios, they are best avoided in loosely-coupled wide-area systems, because adding such state and logic significantly increases system complexity, complicates failure and consistency issues, and creates deployment hurdles. Instead, we want to reuse common, off-the-shelf network substrates for dissemination, because at large scales, such substrates are more robust and less likely to face deployment and maintenance hurdles than more complex networks.

Contributions We provide a complete solution to supporting a large number of select-join subscriptions in a publish/subscribe system. Our goals are to reduce and/or bound bandwidth consumption, notification delay, and server processing costs. Instead of designing a complex broker network, we base our solution on the simple, well-established type of dissemination substrates introduced in Chapter 2, called content-driven networks (CN). We develop novel server-side processing algorithms and application-agnostic extensions to stateless CN to support stateful select-joins and to improve efficiency. More specifically, our contributions include:

- In Section 5.3.1, we propose a method to eliminate result representation and current-content redundancies by rewriting select-join subscriptions into select-semijoins. We demonstrate how these subscriptions can be efficiently processed at the server.

- In Section 5.3.2, we show how to further apply a novel reformulation scheme to eliminate inter-subscription redundancy and enable the use of CN. The reformulation can be computed efficiently at the server, and significantly reduces network traffic.

- In Section 5.5, we propose a simple and novel extension to CN, called *CN**, which further improves efficiency by reducing inter-event and, more generally, re-dissemination redundancies.

- Our solution is general. We show how to handle a generic mix of multi-way select-joins with different sets of joining tables and range selection conditions (Section 5.4). The approach is to decompose such subscriptions into two-way select-semijoins and group-process semijoins

of the same form; we show how to choose a good decomposition using cost-based optimization. We also extend our schemes to support multi-attribute range selections (Section 5.6).

- We have conducted comprehensive experimental evaluation of our solution and its competitors. We measure the performance of server-side processing and dissemination in a simulated wide-area network. Results show orders-of-magnitude improvement for several relevant metrics. We also include examinations of multi-way select-joins and the effectiveness of CN*.

Finally, we note that some of our contributions have applications beyond handling join subscriptions. Interestingly, even for simple selection subscriptions, if events contain bulky payloads (e.g., a PNG image attribute in a `Stocks` event containing the company logo, or a text attribute in `Review` carrying the content of the analyst report), it would be more efficient to treat selections as select-joins and apply our techniques. We discuss this technique using an example in Section 5.6.

The techniques in Section 5.3.1 can be used for delivering results of continuous queries to remote clients, or maintaining views that are materialized remotely, such that communication is minimized. Compared with classic solutions, one advantage of our techniques is they are designed to scale to huge number of continuous select-join queries and views. The techniques in Sections 5.3.2 and Section 5.5 further improve efficiency if we use a broker network for communication.

CN*, the extension to CN, is also a contribution in its own right, as it can be applied to any type of subscription to improve efficiency by optimizing the transfer of payloads between brokers. Again, this extension has been designed with practicality in mind. It is application-agnostic, avoids hard state at brokers, and can be deployed incrementally—i.e., CN* can coexist with CN, and we can gradually replace CN brokers in an existing broker network with CN* brokers, or simply add new CN* brokers to the mix.

5.2 Preliminaries

5.2.1 The Publish/Subscribe Model

Traditional publish/subscribe systems assume that events follow some schema, and subscriptions are simple stateless predicates over individual events (e.g., `PER` $\in [45, 70]$ for a `Stocks` event). As discussed in Section 2.2, we use a publish/subscribe model that supports more powerful database-style subscriptions. We assume that a central server maintain a database. Publishers of data send events to the server. Events are modifications (inserts, deletes, and updates) to the database. A subscription is a query Q over the database. On an incoming event, the database state changes from $\mathcal{D}$ to $\mathcal{D}'$. If $Q(\mathcal{D}') \neq Q(\mathcal{D})$, and we say that the subscription Q is *affected* by the event; the task of the publish/subscribe system is to deliver a notification message to Q's subscriber, such that $Q(D')$ can be computed from $Q(D)$ and the content of this notification message.

Recall from Section 5.1 that we use a network of brokers to share the costs of notification delivery across subscriptions. Each broker is responsible for a subset of the subscriptions. The brokers collectively forward notification messages among themselves and to the subscribers.

5.2.2 Content-Driven Networks

A popular technique for building the broker network is to use a content-driven network (CN), a term that we introduced in Section 2.2 to refer to a class of overlay networks for data dissemination. Recall that CN has a clean and simple dissemination interface. Each message contains a list of attribute-value pairs. A subscription is a filter over individual messages, defined as a predicate involving message attributes. CN efficiently routes each message to all matching subscriptions. We can treat CN as a black-box delivery mechanism (refer to Section 2.2 for a discussion of several concrete instances of CN).

While the simplicity of CN's network interface enables efficient and scalable implementations, subscriptions directly supported by CN are limited to predicates (usually selections) over individual event tuples. Thus, CN cannot be directly used for select-joins, whose processing requires informa-

tion beyond individual messages.

5.2.3 Select-Join Subscriptions

In this chapter, we consider subscriptions expressed as orthogonal range selections over natural joins (*select-joins* for short). These subscriptions constitute a significant portion of workloads in practice. The database schema can be modeled as an undirected *join graph*, which may contain cycles. Nodes in this graph represent tables, and edges represent possible joins between tables. Each select-join subscription corresponds to a connected subgraph of the join graph, which we refer to as the *join signature* of the subscription. In addition to the join conditions implied by the join signature, each subscription can specify a local selection for each table, in the form of an orthogonal range condition. In general, such a condition can involve multiple attributes, constrained by different ranges. We assume that subscriptions return all attributes in the result (i.e., no projections), and there are no self-joins.

We will start with a simple scenario, formalized in the example below, where all subscriptions have the same two-way join signature and one single-attribute range selection for each table. In Section 5.4, we show how to extend our solution in Section 5.3 to a general mix of multi-way joins. The extension to multi-attribute range selections is covered in Section 5.6.

Example 6 (Two-Way Select-Joins)**.** *Let $\mathcal{X} = \{X_1, X_2, \ldots\}$ be a set of subscriptions over tables $R(A, B, P_R)$ and $S(B, C, P_S)$. B is the join attribute, A and C are local selection attributes, and P_R and P_S represent all remaining attributes in the respective tables, which we call* payloads*. Each X_i is defined by query*

$$\left(\sigma_{A \in I_i^A} R\right) \bowtie_B \left(\sigma_{C \in I_i^C} S\right),$$

where $I_i^A = [a_i^{\text{low}}, a_i^{\text{high}}]$ and $I_i^C = [c_i^{\text{low}}, c_i^{\text{high}}]$ specify the ranges of X_i's local selection conditions on $R.A$ and $S.C$, respectively. Note that Example 5 fits in this scenario.

5.2.4 Simple Solutions

We now present several simple solutions that can be used in existing systems to support select-join subscriptions. The parallels between these solutions and prior related work is elaborated in Section 5.8.

Enumeration of Direct Notifications (Enum-J) Given an incoming event, one option is to process all subscriptions at the server, compute a notification message for each affected subscription, and unicast this message to the corresponding subscriber. A notification message contains the relational difference between new and old subscription contents, which, in the case of an insertion event, is the result of the select-join evaluated over the inserted tuple and the joining tables. Many processing techniques have been developed, e.g., NiagaraCQ [CDTW00] in the context of continuous query systems.

This scheme corresponds to the naive approach introduced in Section 5.1. Enum-J suffers from large server output size and lack of sharing of dissemination costs. These problems make the approach difficult to scale to a large number of subscriptions.

Relaxation into Single-Table Selections (Rel-Sel) An n-way select-join subscription can be relaxed into n selection subscriptions, one for each joining table. In Example 6, X_i can be relaxed into two subscriptions, $\sigma_{A \in I_i^A} R$ and $\sigma_{C \in I_i^C} S$. All resulting selection subscriptions are handled by CN. R and S events are directly injected into CN, which then routes each event to all matching selection subscriptions. The subscription client maintains the contents of selection subscriptions, and joins them to compute the original subscription.

Rel-Sel avoids result representation redundancy, and its use of CN alleviates inter-subscription redundancy. Rel-Sel may seem to avoid re-dissemination redundancy as well, but it is only because Rel-Sel may transmit far more data than necessary. Client for X_i in fact receives enough data to be able to maintain the cross product $\sigma_{A \in I_i^A} R \times \sigma_{C \in I_i^C} S$, even though only the join is necessary. Losing the filtering power of join conditions can result in much higher traffic due to transmission of unnecessary content to subscribers.

Reformulation as Selections over Join (Ref-J and Ref-J$^+$) Reformulation (see Chapter 2) can support stateful subscriptions over the stateless CN dissemination interface. Recall from Chapter 2 that on a high level, reformulation works as follows. Instead of injecting an event directly into CN, the server reformulates it into one or more messages with attribute-value pairs carrying additional information computed by the server. On the other hand, each subscription is reformulated as a filter over the reformulated messages. CN automatically routes reformulated messages to matching filters (reformulated subscriptions). The reformulated messages, computed by the server with full access to the database and subscription definitions, carry the state necessary for processing the otherwise stateful subscriptions.

We can use a simple reformulation scheme, which we call Ref-J, to support all select-joins with the same join signature. Given an incoming event, the server computes the change to the result of the natural join corresponding to the join signature (with no selections). This change is represented by a set of join result tuples. In the case of an insertion event, for example, these are the result of joining the inserted tuple with the other tables. Then, each join result tuple is injected into CN as a message. Each select-join subscription, on the other hand, is reformulated into a predicate over join result messages according to the subscription's local selection conditions. For instance, in Example 6, the reformulated messages have the format $\langle \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{P_R}, \mathtt{P_S} \rangle$, while each subscription X_i is simply reformulated into the predicate $\mathtt{A} \in I_i^A \wedge \mathtt{C} \in I_i^C$.

Again, CN helps us avoid inter-subscription redundancy. However, result representation redundancy still remains; the content of the incoming event is repeated in every reformulated message that the event generates.[1] Also, Ref-J does not avoid current-content, or more generally, inter-event and re-dissemination redundancies.

Another inefficiency with Ref-J, caused by delayed evaluation of the selection conditions (in CN instead of at the server), is that Ref-J may send out join result tuples that are not interesting to any subscriptions. This problem can be solved by checking each outgoing join result tuple to

[1] Result representation redundancy could be overcome by requiring brokers to batch messages and compress outgoing message batches during message forwarding, but this requires modifying CN to add complex application-specific logic, which we want to avoid.

ensure that at least one subscription would be interested in it. However, this extension, which we call Ref-J$^+$, increases server processing cost, and still inherits all other problems of Ref-J.

5.3 Binary Select-Joins

We now present our solutions for the scenario described in Example 6, where all subscriptions are binary select-joins between R and S. We will show how to generalize our techniques to multi-way joins in Section 5.4. As a first step, in Section 5.3.1, we ignore the dissemination aspect, and focus on how to efficiently compute, at the server, the minimal information to send to each subscription for every event. Next, in Section 5.3.2, we show how to retool the solution in Section 5.3.1 to further leverage CN for efficient dissemination.

5.3.1 Enum-SJ: Towards a Semijoin Approach

We saw in Section 5.1 that two important sources of inefficiency are result representation and current-content redundancies. It turns out that both can be eliminated by decomposing a select-join subscription $X_i = \sigma_{A \in I_i^A} R \bowtie_B \sigma_{C \in I_i^C} S$ into two select-semijoins:

$$X_i^R = \sigma_{A \in I_i^A} R \ltimes_B \sigma_{C \in I_i^C} S, \text{ and } X_i^S = \sigma_{C \in I_i^C} S \ltimes_B \sigma_{A \in I_i^A} R.$$

We call them ***R-semijoin*** and ***S-semijoin*** respectively. Semijoins are a well-known method in distributed databases [BGW$^+$81] for reducing communication when joining across different machines. A semijoin $R \ltimes S$ can be regarded as a generalized filter on R, which returns only those R tuples that join with at least one S tuple.

The insertion of a tuple $t_R = \langle a, b, p_r \rangle$ into table R can affect both R-semijoin and S-semijoin subscriptions. First, the server has to send t_R to every R-semijoin X_i^R where $a \in I_i^A$ and there exists at least one joining S tuple $\langle b, c, p_s \rangle$ such that $c \in I_i^B$. We call these subscriptions ***t_R-affected***, or simply (when clear from the context), ***affected R-semijoins***. Second, for each S tuple $t_S = \langle b, c, p_s \rangle$ that joins with t_R, the server has to send t_S to every S-semijoin X_i^S where $a \in I_i^A$, $c \in I_i^C$, and X_i^S does not already contain t_S (i.e., before t_R is inserted into R, $t_S \notin \sigma_{C \in I_i^C} S \ltimes_B \sigma_{A \in I_i^A} R$). We call these subscriptions ***(t_R-)affected S-semijoins***.

The client for subscription X_i maintains the contents of both X_i^R and X_i^S. Upon receiving notifications to X_i^R or X_i^S, it updates the result of $X_i^R \bowtie X_i^S$, which is equivalent to the original X_i.

As a concrete example, consider the subscription X_1 in Example 5 and the tables showing the current contents of `Stocks` and `Reviews` in Section 5.1. Upon the insertion of a new `Stocks` tuple $s_4 = \langle \text{GOOG}, 52.1, \ldots \rangle$, recall that the naive approach (Enum-J) has to send $s_4 r_2, s_4 r_3, \ldots, s_4 r_{20}$ to X_1. With the decomposition, however, we only need to send s_4 to $X_1^{\texttt{Stocks}}$. Note how semijoin avoids the multiplicity caused by join, thereby eliminating result representation redundancy; only one copy of s_4 is sent no matter how many joining `Reviews` tuples there are. Furthermore, note that nothing needs to be sent to $X_1^{\texttt{Reviews}}$, because the `Reviews` tuples that join with s_4 and satisfy X_1's selection condition on rating, namely $r_2, \ldots, r_{20}$, are already in $X_1^{\texttt{Reviews}}$. Hence, current-content redundancy is also avoided. This example highlights the fact that having an affected R-semijoin does not imply the corresponding S-semijoin is affected as well.

It is not trivial to compute the changes to a large number of select-semijoin subscriptions, since every subscription has different selection conditions. It is especially tricky to decide which S-semijoins need to receive a joining S tuple, because the decision involves testing whether they already contain the tuple. The remainder of this section is devoted to the details of scalable maintenance of R-semijoins and S-semijoins on an insertion into table R. Insertion into S is symmetric; updates and deletions involve straightforward extensions, which we omit for brevity.

Once the changes are computed, we assume (for now) that the server unicasts them to each subscription. We call this scheme Enum-SJ. We will see how to use CN to avoid inter-subscription redundancy and further reduce output size in Section 5.3.2.

Computing Changes to R-Semijoins

On an insertion $t_R = \langle a, b, p_r \rangle$ into R, we need to identify all affected R-semijoins. Several techniques exist in the continuous query processing literature, e.g. NiagaraCQ [CDTW00]. For example, we can first find all subscriptions whose local selection conditions on $R.A$ are satisfied by t_R, which can be done efficiently with an interval tree indexing all I_i^A intervals. For each such

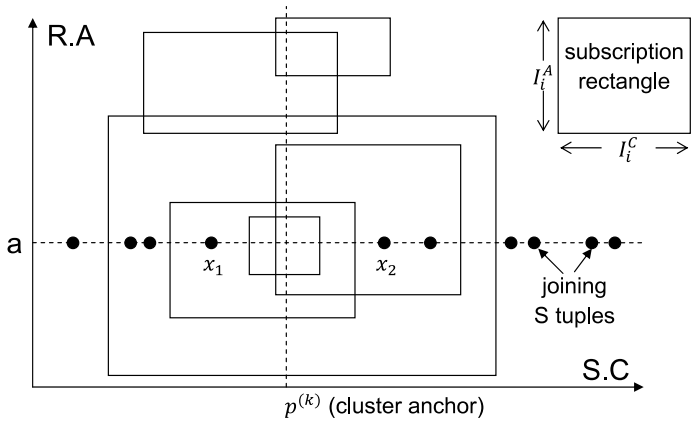


Figure 5.1: Processing Enum-SJ using SSI for cluster $\mathcal{X}^{(k)}$.

subscription X_i, we then probe a composite-key B-tree index on $S(B, C)$ to determine whether there exists at least one S tuple with $B = b$ and $C \in I_i^C$; if yes, the subscription is affected. The problem with this approach is that its running time is linear in the number of subscriptions whose selection conditions on $R.A$ are satisfied, which can be much higher than the actual number of affected subscriptions.

Processing with Stabbing-Set Index (SSI) We base our solution on an algorithm proposed in [AXYY06], which is especially suited to our setting because it exploits the clustering among local selection ranges for efficient processing. In publish/subscribe systems, we expect a fair degree of clustering among subscription ranges because users often share similar interests.

The algorithm is based on a *stabbing-set index* (*SSI*), which partitions the set of subscriptions $\mathcal{X}$ into τ *clusters* $\mathcal{X}^{(1)}, \ldots, \mathcal{X}^{(\tau)}$ according to their I_i^C's, i.e., their local selection ranges on $S.C$. All subscriptions in the k-th cluster $\mathcal{X}^{(k)}$ must have their I_i^C's stabbed by a common point $p^{(k)}$, called the *cluster anchor*. With clustering of user interests, we should be able to partition $\mathcal{X}$ into a relatively smaller number of clusters (i.e., $\tau \ll |\mathcal{X}|$). Efficient algorithms have been developed in [AXYY06] to maintain the partitions such that τ is close to the minimum possible. Figure 5.1 illustrates the set of subscriptions in cluster $\mathcal{X}^{(k)}$ in the space $S.C \times R.A$, where each subscription X_i is represented as a rectangle $I_i^C \times I_i^A$. The rectangles in each cluster are indexed by a 2-d R-tree.

On the insertion of $t_R = \langle a, b, p_r \rangle$ into R, we perform the following steps for each cluster $\mathcal{X}^{(k)}$.

First, by looking up $(b, p^{(k)})$ in the B-tree index on $S(B, C)$, we can find the two joining S tuples with $S.C$ values (say x_1 and x_2) that are the closest possible to $p^{(k)}$ on each side of $p^{(k)}$. Figure 5.1 illustrates this step. Next, we probe the R-tree index for $\mathcal{X}^{(k)}$ to find those subscriptions in $\mathcal{X}^{(k)}$ that contain at least one of the points (x_1, a) and (x_2, a).

For each cluster, this procedure returns exactly all affected R-semijoins within the cluster. To see why, note that we can represent tuples in $\{t_R\} \bowtie S$ as points in the $S.C \times R.A$ space, and all of them must lie on the horizontal line $R.A = a$. An R-semijoin is affected iff its rectangle contains at least one of these points. Therefore, any affected subscription in $\mathcal{X}^{(k)}$ must necessarily contain $(p^{(k)}, a)$; for it to contain a tuple in $\{t_R\} \bowtie S$, it must further contain at least one of (x_1, a) and (x_2, a).

Complexity The space required for auxiliary data structures is $O(|\mathcal{X}| + |S|)$, i.e., linear in the number of subscriptions and the size of the database. Let k_R be the number of affected R-semijoins. Let $g = O(\sqrt{\max_{1 \le k \le \tau} \left|\mathcal{X}^{(k)}\right|})$ denote the cost of a lookup in an R-tree indexing all subscriptions in a cluster (excluding the cost component that is linear in the output size of the lookup). The running time is $O(\tau(\log |S| + g) + k_R)$. Let h_R be the size of one R tuple. The output size, as measured by the total size of all notification messages, is $O(k_R h_R)$.

Computing Changes to S-Semijoins

Given an insertion $t_R = \langle a, b, p_r \rangle$ into R, we need to send a joining S tuple t_S to an S-semijoin X_i^S iff t_S is ***exposed*** to X_i^S, i.e., t_S should be in X_i^S after the insertion but currently is not. It may seem that we need to examine the content of the S-semijoin to determine whether t_S is exposed to it. Interestingly, we show that exposure testing is not as hard as it seems—a single index lookup suffices.

Theorem 4 (Exposure Test)**.** *Given an insertion t_R into R, let*

- $a^-(t_R) = \max\{t.A \mid t \in R \wedge t.B = t_R.B \wedge t.A \le t_R.A\}$*, or* $-\infty$ *if the input to* max *is empty; and*

- $a^+(t_R) = \min\{t.A \mid t \in R \wedge t.B = t_R.B \wedge t.A \geq t_R.A\}$, *or* $+\infty$ *if the input to* $\min$ *is empty.*

In both definitions, R *refers to the state of* R *before the insertion. Consider any* $t_S \in S$ *that joins with* t_R *(i.e.,* $t_S.B = t_R.B$*):* t_S *is exposed to* X_i^S *iff* $t_S.C \in I_i^C$ *and* $t_R.A \in I_i^A \subset [a^-(t_R), a^+(t_R)]$.

Proof. (Sketch) If $t_R.A \notin I_i^A$, X_i^S is clearly not affected by the insertion. Let $t_R.A \in I_i^A$. There are two cases: 1) If $I_i^A \not\subset [a^-(t_R), a^+(t_R)]$, it must contain one of $a^-(t_R)$ and $a^+(t_R)$. Either way, there already exists $t'_R \in R$ with $t'_R.A \in I^A$ and $t'_R.B = t_R.B$, such that any S tuple that should be in X_i^S due to insertion of t_R must already be in X_i^S because of t'_R. 2) If $I_i^A \subset [a^-(t_R), a^+(t_R)]$, there is no other $t'_R \in R$ with $t'_R.A \in I^A$ and $t'_R.B = t_R.B$. Therefore, any S tuple that should be in X_i^S due to insertion of t_R must not be in X_i^S currently. □

Applying Theorem 4, we can compute changes to S-semijoins as follows. First, we compute $a^-(t_R)$ and $a^+(t_R)$, which can be done efficiently by looking up (b, a) in the B-tree index on $R(B, A)$. Next, we process each subscription cluster $\mathcal{X}^k$ in turn, in the same loop where we compute affected R-semijoins (Section 5.3.1). For each affected R-semijoin X_i^R, if $I_i^A \subset [a^-(t_R), a^+(t_R)]$, the corresponding S-semijoin X_i^S is also affected, and its change includes the set of S tuples with $S.B = b$ and $S.C \in I_i^C$, which can be easily found in a B-tree index on $S(B, C)$. As an optimization, we do not need to repeat this lookup from the B-tree root for every affected S-semijoin. Instead, we use a single lookup of $(b, p^{(k)})$ for each cluster; to find S tuples with $S.B = b$ and $S.C \in I_i^C$ for X_i in this cluster, we simply traverse the B-tree leaves towards left and right starting from this point, stopping when we encounter a tuple with $S.C \notin I_i^C$.

Complexity The total space requirement (including auxiliary data structures in Section 5.3.1) is $O(|\mathcal{X}| + |R| + |S|)$, i.e., linear in the number of subscriptions and the size of the database. Let s be the number of S tuples that join with t_R. Recall that k_R denotes the number of affected R-semijoins. Let k_S ($\leq k_R$) denote the number of ***(t_R-)affected S-semijoins*** (i.e., those with at least one exposed joining S tuple), and let $\bar{s}'$ ($\leq s$) be the average number of joining S tuples exposed

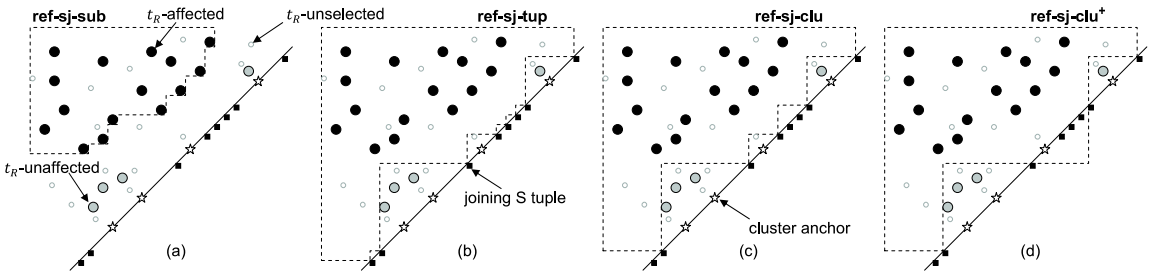


Figure 5.2: Descriptive skylines.

to each affected S-semijoin. The running time, combined with the algorithm in Section 5.3.1, is $O(\log|R| + \tau(\log|S| + g) + k_R + k_S\bar{s}')$. The output size, as measured by the total size of all notification messages for S-semijoins, is $O(k_S\bar{s}'h_S)$, where h_S is the size of one S tuple.

5.3.2 Ref-SJ: Scalable Dissemination

Enum-SJ suffers from inter-subscription redundancy because it unicasts notifications to each subscription. In this section, we present Ref-SJ, which uses novel reformulation techniques to leverage CN for sharing dissemination costs. The overall approach is to have the server reformulate each incoming event into CN messages carrying additional information computed by the server, such that the otherwise stateful subscriptions can be reformulated accordingly into stateless filters supported by CN. Assume as before that we insert an R tuple $t_R = \langle a, b, p_r\rangle$.

Reformulating R-Semijoins

We need to send t_R to the set of affected R-semijoins efficiently. Can we characterize this set succinctly without enumerating the set membership? The answer is yes. To illustrate, let us map each R-semijoin X_i^R with $I_i^C = [c_i^{\text{low}}, c_i^{\text{high}}]$ to a point $(c_i^{\text{low}}, c_i^{\text{high}})$ in a 2-d space, as shown in Figure 5.2 (a).

Consider just the subset of R-semijoins whose I_i^A ranges contain a; we call these the ***(t_R-)selected*** R-semijoins. Obviously, every affected R-semijoin must first be selected. Among the selected R-semijoins, it turns out that the affected ones can be separated from the unaffected ones by a skyline, as stated by Theorem 5 below.

Definition 3. *A* skyline (viewing from northwest) *in* 2*-d is specified by a set of* skyline points $\{(x_1, y_1), (x_2, y_2), \ldots\}$ *with the property that no point lies to the northwest of another point; i.e., there exist no* i, j *such that* $x_i \leq x_j$ *and* $y_i \geq y_j$*. The* subspace covered *by this skyline is the union of northwest quadrants of all skyline points, i.e.,* $\{(x, y) \mid \exists i : x \leq x_i \wedge y \geq y_i\}$.

Theorem 5 (Descriptive Skyline)**.** *There exists a skyline (viewing from northwest)* L *such that 1) every* t_R*-selected* R*-semijoin in the subspace covered by* L *is* t_R*-affected, and 2) every* R*-semijoin not in this subspace is not* t_R*-affected. We call a skyline satisfying these properties a* descriptive skyline*.*

Proof. (Sketch) Let P be the (duplicate-free) set of points corresponding to all affected R-semijoins. Let P' be a subset of P, such that $p \in P'$ iff p does not lie to the northwest of any other point in P. The skyline specified by P' satisfies both properties. 1) If a selected R-semijoin X_i^R is covered by the skyline, it must lie to the northwest of some affected R-semijoin X_j^R, which implies that $I_i^C \supseteq I_j^C$. Therefore, any joining S tuple that satisfies X_j^R's selection condition must satisfy X_i^R's too. Hence, X_i^R must also be affected. 2) This property follows from the construction of P'. □

For example, Figure 5.2 (a) shows one such skyline with 7 skyline points (as constructed by the proof of Theorem 5). Using Theorem 5, our approach is to first compute a descriptive skyline at the server, given $t_R = \langle a, b, p_R \rangle$. Suppose this skyline has k_L points $(x_1, y_1), \ldots, (x_{k_L}, y_{k_L})$. We reformulate t_R into the message:

$$\langle \mathtt{A}{:}a, \mathtt{B}{:}b, \mathtt{P_R}{:}p_r, \mathtt{X}_1{:}x_1, \mathtt{Y}_1{:}y_1, \ldots, \mathtt{X}_{k_L}{:}x_{k_L}, \mathtt{Y}_{k_L}{:}y_{k_L} \rangle.$$

Accordingly, an R-semijoin is reformulated as a filter over this message, which now can directly be handled by stateless CN:

$$\mathtt{A} \in I_i^A \wedge (\exists j : [\mathtt{X}_j, \mathtt{Y}_j] \subseteq I_i^C).$$

Compared with the approach in Section 5.3.1, which sends out k_R unicast messages with a total size of $O(k_R h_R)$, this approach sends out a single CN message with size $O(h_R + k_L)$, where k_L, the number of points in the descriptive skyline, can be potentially much smaller than k_R, the number of points covered by the skyline.

Note that in general there can be many possible descriptive skylines. We now examine several

techniques for computing one. Several factors influence our choice. We want to: 1) compute the skyline efficiently; 2) reduce the number of skyline points, because it affects the reformulated message size; 3) reduce the area of the subspace covered by the skyline, because a larger area may cause slightly higher dissemination costs for some CN implementations.

Minimum-Area Skyline (Ref-SJ-Sub) This is the skyline constructed in the proof of Theorem 5. While it minimizes the area, it may still have a large number of points. Furthermore, computing this skyline requires additional $O(k_R \log k_R)$ time after identifying all k_R points, which is not very desirable.

Joining-Tuple Skyline (Ref-SJ-Tup) Suppose there are s joining S tuples, with $S.C$ values $c^1, \ldots, c^s$. Interestingly, the skyline with points $(c^1, c^1), \ldots, (c^s, c^s)$ is also a descriptive skyline, as illustrated by Figure 5.2 (b). While very easy to compute, the number of skyline points may be large and can even exceed k_R.

Cluster-Based Skyline (Ref-SJ-Clu) Here, as in Section 5.3.1, we again leverage the SSI to exploit the clustering among subscription ranges. We start with an empty point set P. For each cluster $\mathcal{X}^{(k)}$, we look for the two joining S tuples with $S.C$ values (say x_1 and x_2) that are the closest possible to $p^{(k)}$ on each side of $p^{(k)}$. They can be found by looking up $(b, p^{(k)})$ in the B-tree index on $S(B, C)$. We add (x_1, x_1) and (x_2, x_2) to P. Intuitively, the set of affected R-semijoins in this cluster is exactly the set of selected R-semijoins in this cluster that lie to the northwest of (x_1, x_1) or (x_2, x_2). The justification (omitted for brevity) follows the same line of reasoning as the SSI-based algorithm in Section 5.3.1. Briefly, the other joining S tuples which were a part of the description in Ref-SJ-Tup are unnecessary. To see why, note that every t_R-affected R-semijoin is known to lie to the northwest of some cluster (by definition of a cluster), as well as to the northwest of some point in the joining-tuple skyline. Thus, every t_R-affected R-semijoin would be covered by (i.e., lie to the northwest of) at least one of (x_1, x_1) and (x_2, x_2).

When all clusters have been processed, the point set P specifies a descriptive skyline with up to 2τ points, as illustrated by Figure 5.2(c). This descriptive skyline can be quite succinct for

workloads that exhibit a high degree of subscription clustering. The running time of the algorithm is $O(\tau \log |S|)$.

Improved Cluster-Based Skyline (Ref-SJ-Clu$^+$) The skyline of Ref-SJ-Clu can be further compressed. Let $(q_1, q_1), \ldots, (q_n, q_n)$ be a contiguous subsequence of skyline points, sorted from southwest to northeast. Consider the sawtooth region between the skyline and the diagonal line segment from (q_1, q_1) to (q_n, q_n). If this sawtooth region contains no R-semijoin that is both selected and unaffected, we can replace the subsequence of skyline points by a single point (q_n, q_1) and obtain a simpler descriptive skyline, as illustrated by Figure 5.2 (d).

This improvement, which we call Ref-SJ-Clu$^+$, can be realized algorithmically as follows. We generate the Ref-SJ-Clu skyline points in order, by processing clusters in the increasing order of their anchors. During this process, we check, for each pair of consecutive skyline points (q_j, q_j) and (q_{j+1}, q_{j+1}), whether the triangle they form together with (q_j, q_{j+1}) is *disposable*, i.e., contains no R-semijoin that is both selected and unaffected. Once we identify a maximal consecutive sequence of at least two disposable triangles, we can replace the corresponding skyline points by a single one.

We can locate all disposable triangles using at most τ R-tree lookups. Specifically, we check whether a triangle between a pair of consecutive skyline points (q_j, q_j) and (q_{j+1}, q_{j+1}) is disposable as follows. If there is no cluster anchor between q_j and q_{j+1}, the triangle cannot contain any subscription at all (otherwise this subscription would not belong to any cluster); therefore, the triangle is obviously disposable. Suppose there are one or more cluster anchors between q_j and q_{j+1}. For each such cluster anchor $p^{(k)}$, we look up $(p^{(k)}, a)$ in the cluster R-tree. For each t_R-selected R-semijoin returned by the lookup, we check if that semijoin is also t_R-affected, by testing whether it contains either (q_j, a) or (q_{j+1}, a). As soon as we encounter a semijoin that is not t_R-affected, we can terminate the process immediately for that triangle, and declare that triangle to be not disposable. Otherwise, the triangle is reported as disposable.

Because of the above termination condition, for each group, we examine at most one semijoin that is not t_R-affected; all other semijoins we examine are t_R-affected. Therefore, the total running time for locating all disposable triangles is $O(\tau g + k_R)$, where k_R is the total number of t_R-affected semijoins.

The number of Ref-SJ-Clu$^+$ skyline points is still 2τ in the worst case, but in practice we find the number to be much lower. Although the algorithm by itself is more costly than Ref-SJ-Clu, it can be combined with the algorithm for computing reformulated messages for S-semijoins (Section 5.3.2) without increasing the overall asymptotic complexity.

Finally, we note that even Ref-SJ-Clu$^+$ may not find a descriptive skyline with the minimum number of points. It is possible to find such a skyline, but the running time would become $O(k_R^2)$. We leave a more detailed investigation of this alternative as future work, since our experiments show that Ref-SJ-Clu$^+$ works well in practice and offers very good compression at low cost.

Reformulating S-Semijoins

Given an insertion t_R, our overall strategy is to first find the set of S tuples that are exposed to at least one S-semijoin; for each such S tuple, we create a CN message containing its content plus some additional information so that CN can route it to the exact set of S-semijoins to which it is exposed.

The algorithm proceeds as follows. Given $t_R = \langle a, b, p_r \rangle$, we first compute $a^-(t_R)$ and $a^+(t_R)$ by looking up (b, a) in the B-tree index on $R(B, A)$, as in Section 5.3.1. Next, we compute $\mathcal{I}$, the union of I_i^C ranges of all affected S-semijoins, by iterating through subscription clusters. For each cluster $\mathcal{X}^{(k)}$, we find the affected R-semijoins using two R-tree lookups, as in Section 5.3.1. For every affected R-semijoin X_i^R with $I_i^A \subset [a^-(t_R), a^+(t_R)]$, we add its I_i^C range to $\mathcal{I}$. During this process, we maintain $\mathcal{I}$ as a sequence of maximal, non-overlapping ranges. Finally, for each range I in $\mathcal{I}$, we retrieve all S tuples with $S.B = b$ and $S.C \in I$, using a B-tree index on $S(B, C)$. For each retrieved S tuple t_S, we inject the following reformulated CN message:

$$\langle \mathtt{B}{:}b, \mathtt{C}{:}t_S.C, \mathtt{P_S}{:}t_S.p_s, \mathtt{A}{:}a, \mathtt{A}^-{:}a^-(t_R), \mathtt{A}^+{:}a^+(t_R) \rangle.$$

An S-semijoin is reformulated as a simple filter over this message:

$$\mathtt{A} \in I_i^A \subset [\mathtt{A}^-, \mathtt{A}^+] \wedge \mathtt{C} \in I_i^C.$$

For one incoming event, the number of notification messages is s', the number of S tuples exposed to at least one affected S-semijoin. The total message size is $O(s'h_S)$. The running time of the algorithm is $O(\log |R| + \tau(\log |S| + g) + k_R \log k_S + k_S \log |S| + s')$.

Recap and Comparison with Enum-SJ

To recap, Ref-SJ operates as follows. When a select-join subscription is created, it is decomposed into R- and S-semijoins, which are then reformulated into message filters (as described in Sections 5.3.2 and 5.3.2, respectively) and registered in the CN.

Given an incoming insertion t_R into R, the server generates a message containing t_R together with the descriptive skyline (Section 5.3.2). When injected into CN, this message is automatically routed by CN to reach all affected R-semijoins. Also, the server generates a series of messages, one for each S tuple that is exposed to least one S-semijoin (Section 5.3.2). Every message is augmented with $t_R.A$, $a^-(t_R)$, and $a^+(t_R)$, and is automatically routed by CN to reach any affected S-semijoin that it is exposed to.

Compared with Enum-SJ, not only does Ref-SJ leverage CN for efficient dissemination, but Ref-SJ also makes it possible to speed up server processing. Intuitively, the complexity of Enum-SJ, as any method that enumerates all affected subscriptions, is fundamentally lower-bounded by the number of affected subscriptions. Reformulation-based approaches, on the other hand, only need to generate a description of this set, which can be much more concise.

5.4 Multi-Way Select-Joins

In this section, we consider how to handle a general mix of select-join queries over multiple tables. Recall from Section 5.2 that we represent the schema as a join graph, and each select-join query has a join signature that corresponds to a connected subgraph of the join graph. The join graph has tables as nodes and join attributes as edges. For simplicity, we assume that each table has a single selection attribute, and defer the extension to multiple attributes to Section 5.6.3. Each table has as many join attributes as incident edges. We present our proposed approach in this section, and defer a discussion of other alternatives and the associated tradeoffs to Section 5.6.4.

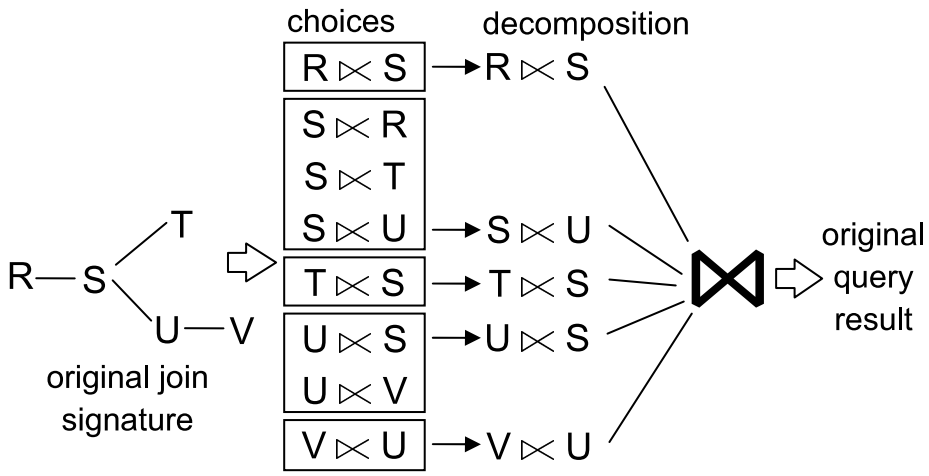


Figure 5.3: Processing multi-way join using decomposition.

5.4.1 Overview

Our approach decomposes each n-way select-join over n input tables into n binary select-semijoins (exactly one for each input table). The select-semijoin for input table R is the semijoin of R with one of the neighboring input tables of R (say S) in the join signature, i.e., $\sigma_{p_R} R \ltimes_B \sigma_{p_S} S$, where B is the common join attribute, p_R and p_S refer to the selection conditions on R and S in the original subscription. We call this binary select-semijoin an R^S-*semijoin*; here, R^S denotes the *form* of the semijoin, where R is called the *base table* and S is called the *filtering table*. An R^S-semijoin is identical in definition to the R-semijoin introduced in the two-way case; the superscript S serves to disambiguate between the various possible neighboring tables in the join graph that R could join with. For example, the five-way select-join whose signature is shown in Figure 5.3 might be decomposed into 5 semijoins: R^S, S^U, T^S, U^S, and V^U. Note that there are three choices of filtering table for base table S, corresponding to the three edges incident to S. Likewise, there are two choices for table U—U^S-semijoin and U^V-semijoin.

Instead of the original multi-way select-join subscription, the subscription client would maintain the decomposed binary select-semijoins. The client can reconstruct the original select-join using these binary select-semijoins by simply performing an n-way natural join over the contents. To see why, note that the client receives input table tuples that participate in some two-way select-join

result. Since we know that every multi-way select-join result has an embedded two-way select-join result, the destination can use the semijoin decomposition to answer the original select-join subscription. Note that no local selection (filtering) is necessary at the client, because the binary semijoins already handle these predicates. The process of decomposing a join signature and reconstructing the original query result using the binary semijoins is shown in Figure 5.3.

One consequence of binary semijoin decomposition is that not every semi-join result is necessarily a part of the multi-way select-join result, which means that the query could receive some false positives. Nevertheless, in cases with many join results (which is where simpler methods suffer the most) the number of tuples passing the binary semijoin is usually much smaller. We consider the alternative of directly extending semijoins (without decomposition) to the multi-way case, in Section 5.6.4.

Using the techniques from Section 5.3, we process the decomposed binary select-semijoins in groups, where each group contains all binary select-semijoins of the same form.

5.4.2 Optimizing Decomposition

Given a set of multi-way select-joins (with different signatures in general), we are faced with the following optimization problem. How should we decompose each of these multi-way select-joins such that the resulting groups of binary select-semijoin are least expensive to process? We refer to the decomposition decisions we make on the given set of select-joins collectively as a *decomposition* for this set.

For some intuition, suppose we want to choose a binary select-semijoin for a base table R in a multi-way select-join. Consider the implications of choosing a particular filtering table S. 1) On the insertion of tuple t_R into table R, t_R has to be sent to all t_R-affected R^S-semijoins. The associated cost depends on the probability of insertion into table t_R and the expected number of t_R-affected R^S-semijoins. Intuitively, we want to choose a "selective" filtering table S that is expected to lead to fewer t_R-affected R^S-semijoins. 2) On insertion of a tuple t_S into table S, each of the joining R tuples needs to be sent to all R^S-semijoins to which it is exposed. The associated cost depends on the probability of insertion into table t_S, the expected number of joining R tuples, and the expected

number of t_S-affected R^S-semijoins. Intuitively, we again prefer to choose a "selective" neighbor S whose updates impact R^S-semijoins minimally. Note that choosing R^S-semijoin does not imply a preference for choosing S^R-semijoin, i.e., the choice of filtering table for each base table need not be reciprocal.

We cost a decomposition as follows. The total cost is the sum over all resulting *semijoin groups*. A semijoin group (say R^S) contains all decomposed binary select-semijoins of the same form (R^S-semijoins). The cost of the R^S group is the number of R^S-semijoins assigned to the group times a per-semijoin cost $c(R^S)$. We propose two techniques for estimating the per-semijoin cost for a group: One is based on periodic simulation of samples of subscriptions and events, and the other one is based on a simple parametric cost model (see Section 5.4.3 for details).

Given a set of multi-way select-joins whose signatures are drawn from a join graph, we use a greedy algorithm to find the decomposition with the lowest cost. We repeat the following for each node R in the join graph. We find the neighbor S with the lowest $c(R^S)$, and choose R^S-semijoins for all subscriptions involving both R and S. If there is any subscription involving R for which we have not yet chosen a semijoin for R, we repeat the process using the neighbor S' with the next lowest per-semijoin cost $c(R^{S'})$. After a choice for base table R has been made for all subscriptions involving R, we move on to another node in the join graph.

Under the assumptions of our cost function, it is easy to see that this greedy algorithm finds the optimal decomposition (see below for a proof). In Section 5.7, we shall see that optimization based on the simple parametric cost model gives good results.

Optimality of Greedy The greedy decomposition gives the optimal choice under our cost model. We prove this using a cut-and-paste argument as follows. If some table R were to select a binary semijoin R^S-semijoin with a per-semijoin cost c_1, where c_1 is not the minimum, i.e., c_1 is greater than the per-semijoin cost c_2 of some other binary semijoin (say R^T-semijoin) for R, then by selecting R^T-semijoin with per-semijoin cost c_2 instead of R^S-semijoin for table R, we would be able to reduce the total cost of the decomposition by $(c_1 - c_2)$, thus proving that the original choice of R^S-semijoin (with cost c_1) was suboptimal.

Complexity Let e denote the number of edges in the join graph. The time complexity of the greedy algorithm is $O(e(\log e + |\mathcal{X}|))$, dominated by outputting the result decomposition. The part of the running time attributed to decision making is only $O(e \log e)$.

The total space required by all our auxiliary data structures is $O(\bar{n}\,|\mathcal{X}| + \sum_i e_i\,|T_i|)$, where $\bar{n}$ is the average number of tables in a multi-way join, and e_i is the number of edges incident to T_i in the join graph. Basically, each n-way select-join contributes n binary select-semijoins. For each group of binary select-semijoins, we need two SSIs—one for each of the two local selection attributes. An SSI (including R-trees for its clusters) takes space linear in the number of semijoins in the group. Therefore, the total space taken by SSIs is $O(\bar{n}\,|\mathcal{X}|)$. In addition, for each table, we need one composite-key B-tree for each of its join attributes (in combination with the local selection attribute). These B-trees together take $O(\sum_i e_i\,|T_i|)$ space.

Alternative Approaches There are a number of other approaches to handling multi-way select-joins, such as Rel-Sel and Ref-J (Section 5.2.4), as well as a non-trivial extension of the semijoin approach to longer select-semijoins of the form $R \ltimes (S \bowtie T \bowtie \cdots)$. In particular, it may seem that this last alternative, with its greater filtering power, could be better than our approach of using only binary select-semijoins, because with only binary select-semijoins we might send tuples participating in a local binary select-join but not the original multi-way query. However, the alternatives have problems that make them less attractive than our approach; we refer the interested readers to Section 5.6.4 for details.

5.4.3 Estimating Per-Semijoin Cost

As briefly discussed in Section 5.4.2, we propose two methods for estimating the per-semijoin cost for a semijoin group in a decomposition of a set of multi-way select-joins.

Periodic Simulation We use a random sample of subscriptions and events, and simulate processing and dissemination for each possible group. We let each group R^S include all R^S-semijoins available for choice (only for the purpose of estimation—such assignments do not collectively form

a valid decomposition). The per-group cost obtained from simulation is divided by the size of the group to give a per-semijoin cost for this group. This approach is general and can adapt to the actual subscription and event workloads, but simulation incurs overhead.

Parametric Cost Model To keep model complexity low, we make a number of assumptions and simplifications. Consider the select-semijoin $\sigma_{p_R} R \ltimes_B \sigma_{p_S} S$, with the CAN-style CN introduced in Section 2.2.4. Assume that 1) subscriptions are uniformly distributed in terms of their range selection predicates, and 2) the events' local selection attribute values are uniformly distributed over their respective domains.

With a binary join, the CAN space is four-dimensional. Two dimensions correspond to the left and right endpoints of $R.A$ selection ranges. We call the projection of the CAN space onto these two dimensions the R^2-*space*. The remaining two dimensions of the CAN space correspond to the left and right endpoints of $S.C$ selection ranges, and we call this space the S^2-*space*.

All subscriptions lie in what we call the *routing area* of the CAN space, which is the product of its projections onto the R^2-space and the S^2-space. The projection of the routing area onto the R^2-space (S^2-space) is the area to the upper-left of the diagonal of the R^2-space (S^2-space, respectively). We assume that the routing area is divided by a uniform grid into zones, each of which hosts approximately the same number of subscriptions. The cost of disseminating a message to a region depends on the number of zones covered during routing, and hence is roughly proportional to the fraction of the routing area covered by the region.

Our cost model uses the following parameters:

- p_R, probability that a given event is an insertion into table R;
- p_S, probability that a given event is an insertion into table S;
- j_R, expected number of R tuples having the same join attribute value;
- j_S, expected number of S tuples having the same join attribute value.

The cost of the select-semijoin has two components:

- *Cost due to insertion into R.* On an insertion t_R into R, consider first the projection of the affected region onto the R^2-space. The ratio of the area of the affected region to that of the routing area, in R^2-space, is at least 0 (when $t_R.A$ is one of the two extreme values of its domain) and at most $\frac{1}{2}$ (when $t_R.A$ is right in the middle of its domain). Assuming that $t_R.A$ is uniformly distributed, the expected ratio is $(\int_0^1 (1-x)x\ dx)/\frac{1}{2} = \frac{1}{3}$.

  Next, consider the projection of the affected region onto the S^2-space. The ratio of the area of the affected region to that of the routing area, in S^2-space, is at most $\frac{j_S}{j_S+1}$, achieved when the joining S tuples' local selection attribute values divides its domain into $j_S + 1$ equal intervals. Assuming that the j_S values are drawn uniformly, the expected ratio turns out to be $\frac{j_S}{j_S+2}$, which can be calculated by

$$1 - \frac{\int \cdots \int_G (\sum_{i=1}^{j_S} x_i^2 + (1 - \sum_{i=1}^{j_S} x_i)^2)\ dx_{j_S} \cdots dx_1}{\int \cdots \int_G\ dx_{j_S} \cdots dx_1},$$

  where G is the volume $\{(x_1, \ldots, x_{j_S}) \mid x_1, \ldots, x_{j_S} \geq 0 \wedge \sum_{i=1}^{j_S} x_i \leq 1\}$.

- *Cost due to insertion into S.* On an insertion t_S into S, we need to send j_R messages, one for each joining R tuple. Consider the message for a joining R tuple t_R. In the R^2-space, the expected ratio of the area of the affected region to that of the routing area is $\frac{1}{3}$ as before. In the S^2-space, the affected region is characterized by a rectangle cornered at $(t_S.C, t_S.C)$ and $(c^-(t_S.C), c^+(t_S.C))$, where c^- and c^+ are defined analogously as a^- and a^+ in Section 5.3. Suppose there were j_S existing S tuples with the same join attribute value as t_S. Their C values divide a unit-size domain into $j_S + 1$ intervals with lengths $x_1, \ldots, x_{j_S}$, and $x_{j_S+1} = 1 - \sum_{i=1}^{j_S} x_i$. The new insertion falls into the i-th interval with probability x_i; when that happens, the expected area of the affected region in the S^2-space is $\int_0^{x_i} (x_i - x)x\ dx = x_i^2/6$. Overall, in the S^2-space, the expected ratio of the area of the affected region to that of the routing area turns out to be $\frac{2}{(j_S+2)(j_S+3)}$, computed by

$$\frac{\int \cdots \int_G (\sum_{i=1}^{j_S} x_i^3/6 + (1 - \sum_{i=1}^{j_S} x_i)^3/6)\ dx_{j_S} \cdots dx_1}{\int \cdots \int_G\ dx_{j_S} \cdots dx_1} : \frac{1}{2},$$

  where G is the volume $\{(x_1, \ldots, x_{j_S}) \mid x_1, \ldots, x_{j_S} \geq 0 \wedge \sum_{i=1}^{j_S} x_i \leq 1\}$.

Since insertions into R and S occur with probabilities p_R and p_S respectively, the expected total cost is

$$\alpha \cdot p_R \cdot \frac{1}{3} \cdot \frac{j_S}{j_S + 2} + \beta \cdot p_S \cdot j_R \cdot \frac{1}{3} \cdot \frac{2}{(j_S + 2)(j_S + 3)},$$

where α and β are constants.

5.5 Reducing Re-Dissemination Redundancy

The techniques outlined in Sections 5.3 and 5.4 avoid result representation, current-content, and inter-subscription redundancies which cover a very significant portion of network cost. However, they still cannot avoid re-dissemination redundancy across events and subscriptions. In Example 5, a deletion from `Stocks` could remove the joining reviews from X_1's current content. A future update that brings them back into X_1 would cause the reviews to be sent again from the server. More generally, there is no sharing of content across events, because the updates delivered to a broker are not available for sharing in future. This problem is exacerbated when tables have large payloads, such as inline text, video, etc.

The re-dissemination problem is not specific to join subscriptions, but is universal for any subscriptions requesting events with non-negligible payload; we will see an example in Section 5.6. Hence, we opt for a general, CN-based solution that attempts to avoid retransmitting the same payload through the same link. We aim to maintain the clean interface of CN, and avoid introducing hard state or complex processing at brokers.

One straightforward solution to the problem is caching. The idea is to assign each payload a unique reference, and push messages with the reference, but not the payload, to subscriptions. Upon receiving the message, a subscription client sends a request to the server for the payload with the reference. The request and its reply are routed over an overlay network that implements caching, so as to serve future requests before they reach the server. Unfortunately, this scheme fundamentally changes the push-style dissemination of publish/subscribe to pull (for payloads), which may not be acceptable to some applications. Moreover, cache misses add considerable latency, and new payloads will always result in initial cache misses, potentially causing a high amount of additional

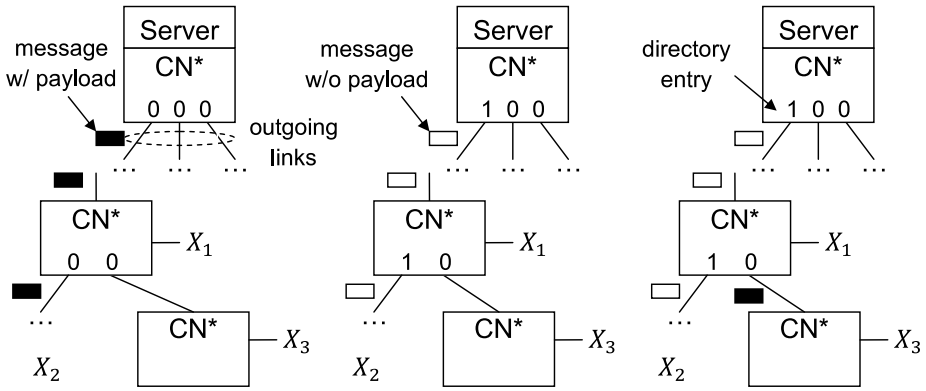


Figure 5.4: Payload dissemination using CN*.

pull traffic.

Overview of CN* We now present our solution called *CN**. In order to avoid sending the payload over a link multiple times, we extend CN to maintain a *payload directory* and a *payload repository* at each node. The server assigns a unique ID for each payload (e.g., using a content hash). The payload repository essentially caches payloads along the push path. The payload directory remembers whether a particular payload has been sent through an outgoing link. Briefly, we avoid re-dissemination as follows: If a payload has been sent over an outgoing link, the node replaces the payload with just the payload ID, and sends this "lighter" version of the message instead; otherwise, the message is sent with the payload. The node responsible for a subscription will replace any payload ID in the message with the actual payload before finally forwarding the message to the subscription client.

Consider the example in Figure 5.4. On the first event, subscriptions X_1 and X_2, both interested in this event, receive the same payload through CN* (Figure 5.4 left). Later, if the same payload is needed again by X_1 and X_2 due to another event (which joins with the same tuple as the first event), CN* will send the message without the payload (Figure 5.4 center). Finally, if another event causes a third subscription X_3 to need the same payload, CN* will transmit the payload only along those links that have not sent it previously; other links will transmit without the payload (Figure 5.4 right).

One important feature of CN* is that both payload directory and repository in a CN* node are

soft state whose size can be capped. We do not assume that each CN* node can remember the entire transmission history. Soft state also aids in failure handling, because we do not have to recover any state related to the payload. However, this means that we must cope with cases where the directory and/or repository entries are deleted. We next describe the details of our approach.

Operational Details A standard CN message, which is a list of attribute-value pairs, is augmented with three attributes with special meanings to CN*. 1) PayloadAttrs specify which attributes in the message collectively form the payload. 2) ID is a unique identifier for the content of the payload in this message. 3) Source records the last encountered CN* node along the dissemination path, which may have a copy of the payload in its cache. If no such CN* node exists (e.g., when the message just enters the CN*), Source is set to the IP address of the server, which is ultimately responsible for supplying the content of the payload when necessary.

At each CN* node, the payload directory entries have the form [ID, Bitmap]. Bitmap is a bitmap with one bit for each outgoing link, which records whether a payload has been previously sent over that link. The payload repository entries have the form [ID, Payload], where Payload stores the actual payload content. Both the directory and the repository are indexed on ID to support quick access to a particular entry.

Assume for now that entries in the directory and repository are never purged. The server first injects the original message with the three additional attributes into CN*. Upon receiving a message m, each CN* node checks its directory to see if an entry for m.ID is present. If not, it creates a new directory entry with m.ID and an empty bitmap. It also adds the content of the payload, if available in m, to its payload repository. Link matching is then done just as in regular CN, to determine the set of outgoing links to which the incoming message needs to be forwarded. For each matching link, if the same payload was previously sent over that link (indicated by a 1 in the directory entry bitmap), the node sends the message without the payload (by removing values for attributes in m.PayloadAttrs). Otherwise, the node sends the message with the payload (reconstructed from the local repository if necessary), and sets the appropriate bit in the directory entry bitmap to 1. In either case, a node that has the payload in its repository always sets Source to its own address before forwarding the message.

Directory and Repository Maintenance To limit space usage, CN* may need to purge entries from the repository and/or directory. The choice of which entry to purge is analogous to a cache replacement policy; we use a least-recently-used (LRU) scheme. Further, if a directory entry is 1 for all outgoing links, it is okay to purge the entry from the repository. We next discuss the handling of purged repository and directory entries separately.

Handling Purged Repository Entries: Assume that a message m without the payload arrives at node n, but there is no repository entry for the payload at n (because it has been purged). If n needs to send the payload along some outgoing link or to a subscription client, n must first obtain the payload by contacting m.Source with m.ID. In this case, the increase in notification latency for subscriptions in n's subtree is the roundtrip time between n and m.Source. Note that if m.Source has dropped the entry in the interim period, n can directly contact the server as a fallback mechanism. This scheme incurs no more payload traffic (one hop) than directly carrying the payload from m.Source. The increase in notification latency for this subtree is one roundtrip between n and m.Source.

Handling Purged Directory Entries: When n receives a message whose payload ID corresponds to a directory entry purged earlier, n simply assumes that all bitmap entries are 0 (as if the payload is new). Thus, with purging of directory entries, n may send the same payload again along a link. However, note that the directory occupies very little space (a 512kB directory can hold tens of thousands of entries), so the chance of purging a useful directory entry is very low in practice, and re-dissemination can be usually avoided.

Forwarding Algorithm Algorithm 4 shows the CN* forwarding procedure for an incoming message m. For simplicity of presentation, we assume that there is only one attribute (Payload) in PayloadAttrs. In Line 3, R-QUERY retrieves the payload directory and repository entries (represented together as X) for m.ID (a new directory entry is created if necessary). If the payload is present in the incoming message m, R-QUERY also adds the payload to the payload repository. Line 4 identifies the matching outgoing network interfaces just as in CN. In Lines 7–11, we check the directory entry bitmap (X.Bitmap) for each affected interface. If the payload was previously sent over that interface, the node strips the payload from the message (Line 8). Otherwise, if m

does not already contain the payload, GETPAYLOAD (Line 10) retrieves the payload (X.Payload) from the payload repository at the closest provider source (m.Source) or the server (in the worst case). The retrieved payload is added to the message (Line 11). If the repository contains the payload, Line 12 updates the Source field in the outgoing message to the machine's network address. Finally, the message is disseminated along the outgoing link (Line 13).

Algorithm 4: Forwarding algorithm for CN*.

```
1  FORWARD(message m) begin
2      p ← ∅ ;                                   // placeholder for payload
3      X ← R-QUERY(m) ;            // query the directory and repository
4      H ← GETMATCHES(m) ;                  // get the matching interfaces
5      foreach interface i in H do
6          m' ← m;
7          if X.Bitmap[i] = 1 then
8              m'.Payload ← ∅ ;                      // payload sent previously
9          else if m.Payload = ∅ then
               // add payload to message
10             if p = ∅ then p = GETPAYLOAD(X, m.Source);
11             m'.Payload ← p;
           // update source address
12         if X.Payload ≠ ∅ then m'.Source ← local address;
13         DISSEMINATE(m', i) ;              // send message along interface i
14 end
```

Co-existence with Regular CN By design, CN* nodes can co-exist with regular CN nodes. This facilitates incremental deployment and adoption. Regular CN nodes simply ignore the additional attributes in the message. Our algorithms operate correctly in the presence of CN nodes. If

a CN node delivers a message m with empty payload to the broker application (for example, because an upstream CN* node assumed that the payload was previously sent), it can simply contact m.Source to retrieve the payload. Another alternative is for CN* to always send the payload if the downstream node is CN.

We have developed an efficient technique for CN* to detect that a next-hop neighbor is a CN node, without adding overhead to the critical path of dissemination. When a payload is sent along an outgoing link i, the corresponding bit in the payload directory entry is not immediately set. Instead, it is set only if this node receives a special acknowledgment from the next hop (indicating that the next-hop node is a CN* node). Thus, in case the next hop is a plain CN node, the bit would never get set and the payload would always get sent along that link. The acknowledgment does not delay message forwarding at the next hop, and therefore does not increase notification latency.

Comparison with Caching CN* differs from traditional caching in two important ways. First, traditional caching applies to values of identifiable objects, and hence must deal with coherency issues when values change. In contrast, CN* caches just values (of payloads), which identify themselves; each different value is a separate cacheable payload that is immutable by definition. Hence, CN* need not worry about cache updates. Second, as discussed earlier in this section, a straightforward caching solution would generate lots of initial cache misses for any new payload, adding considerable notification latency. In contrast, CN* preserves the push-style dissemination of publish/subscribe. Dissemination of a new payload through CN* involves no misses and is identical in communication pattern to dissemination through regular CN.

5.6 Discussion and Extensions

5.6.1 Select with Payload

A stateless selection subscription, supported by traditional publish/subscribe systems, can benefit from being expressed as a join. For example, a subscriber may be interested in receiving a news feed with all detailed product information for products whose ratings fall within some prescribed

range. The event schema may look like $(\mathsf{ID}, \texttt{Rating}, \texttt{Photo}, \ldots)$, where each event reports the new rating and includes other relevant information, such as a picture for the product. The problem with such a stateless subscription is that a new rating event for the same product would have to carry the bulky `Photo` (and other attributes) repeatedly, and must be delivered all the way to interested subscribers. If we represent each subscription as a select-join over two tables $(\mathsf{ID}, \texttt{Rating})$ and $(\mathsf{ID}, \texttt{Photo}, \ldots)$, our techniques can bring two benefits:

- The use of binary semijoin reformulation directly eliminates result representation and current-content redundancies. This happens automatically due to the reformulation scheme, and does not have to be treated as a special case. In the example above, the `Photo` attribute would not be sent to a subscription if the product's previous rating was already within the subscription's range of interest.

- CN^* can provide further benefits by reducing re-dissemination redundancy. In the example, assume that the `Photo` attribute has been previously delivered to some subscription X_1, and later needs to be delivered to some other subscription X_2. Let X_1 and X_2 share some common path in the overlay dissemination network. In this case, the `Photo` attribute would not be re-disseminated on the common path if it was retained by CN^* in some payload repositories.

5.6.2 Multi-Attribute Join Conditions

Multi-attribute equijoin conditions are straightforward to handle, as we can conceptually treat the set of join attributes as a single composite attribute. Consider the binary join between tables R and S with join attributes $(B_1, \ldots, B_m)$. The only change to our algorithms is to use B-trees indexing composite keys $(B_1, \ldots, B_m, A)$ and $(B_1, \ldots, B_m, C)$, instead of (B, A) and (B, C) respectively.

5.6.3 Multi-Attribute Selection Conditions

Suppose the local selection conditions on R and S are conjunctions of d_R and d_S range conditions, respectively. Consider the case of inserting an R tuple t_R (the case of inserting an S tuple is sym-

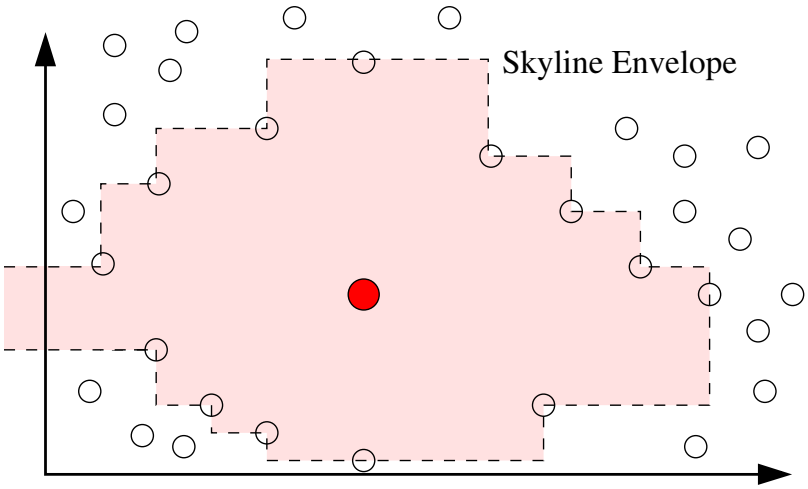


Figure 5.5: Skyline envelope in two dimensions.

metric). We extend our semijoin reformulation techniques in Section 5.3 as follows. A subscription is a hypercube with $d_R + d_S$ dimensions, one for each local selection attribute. The joining S tuples can be mapped as a set of points $\mathcal{J}_S$ in a d_S-dimensional space, which we call the S*-space*, whose dimensions correspond to the local selection attributes of S.

Handling R-Semijoins The reformulation scheme is the same as before, but the descriptive skyline is in a $2d_S$-dimensional space instead of a 2-dimensional space. For each local selection attribute of S, there are two dimensions in this space that correspond to the two endpoints of a range over this attribute. We call this $2d_S$-dimensional space the S^2-space.

Finding a descriptive skyline becomes more complicated in the S^2-space. The joining-tuple skyline (Section 5.3.2) can be directly derived from $\mathcal{J}_S$—each point $(c_1, \ldots, c_{d_S})$ of $\mathcal{J}_S$ in the S-space is converted into a skyline point in the S^2-space by simply repeating each coordinate twice, i.e., $(c_1, c_1, \ldots, c_{d_S}, c_{d_S})$.

We next describe how to derive the cluster-based skyline. First, we introduce the concept of a *skyline envelope*.

Definition 4 (Skyline Envelope)**.** *$Sky(\mathcal{X}, z)$, the* skyline envelope *of point z with respect to a set of points $\mathcal{X}$, is the set of all points $\mathcal{Y} \subseteq \mathcal{X}$ such that for each point $y \in \mathcal{Y}$, no point in $\mathcal{X}$ lies within the minimum hypercube containing both points z and y.*

Figure 5.5 shows an example skyline envelope in two-dimensional space.

For each cluster of subscriptions whose local selections on S are satisfied by a common cluster anchor $p = (c_1, \ldots, c_{d_S})$, we compute $\mathsf{Sky}(\mathcal{J}_S, p)$, i.e., the skyline envelope of p with respect to the joining S tuples in the S-space. Then, for each point in the skyline envelope, we convert this point from the S-space to the S^2-space, again by simply repeating each coordinate twice. The final cluster-based skyline consists of all such points in the S^2-space, obtained from the skyline envelopes of all subscription clusters.

Handling S-Semijoins For each joining S tuple t_S, consider the set of previously joining R tuples $\mathcal{J}_R$ as points in the d_R-dimensional *R-space* (defined analogously as the S-space). We compute $\mathsf{Sky}(\mathcal{J}_R, t_R)$, i.e., the skyline envelope of t_R with respect to $\mathcal{J}_\mathcal{R}$ in the R-space. The points in this skyline envelope are included in the reformulated message for t_S. Each subscription is reformulated to require that its selection conditions are satisfied by t_S and t_R, and that its selection conditions on R are satisfied by none of the points in the skyline envelope. For example, in Figure 5.5, the latter requirement means that the subscription rectangle must lie within the shaded region shown. This latter condition ensures that we do not notify subscriptions whose current contents already contain t_S due to joining with some other selected R tuple.

Discussion While the extension described above is very aggressive in trying to minimize the amount of data to be disseminated, it may be less desirable in practice due to difficulties with high-dimensional indexing and skyline computation, as well as large descriptive skylines and lower degrees of user-interest clustering expected in higher dimensions. Similar to practices in traditional databases, one solution to tackle the problem of higher dimensions is to choose one selection attribute (per table) for group processing and dissemination. The remaining selection conditions are applied by subscribers in a post-processing step. The tradeoffs in choosing the best selection attribute in this case are similar to that of choosing the best binary semijoin decomposition in the multi-way join case (we need to choose the overall most selective and effective filtering attribute for each table). We leave the in-depth study of how to choose the best single selection attribute as future work.

5.6.4 Alternative Approaches for Handling Multi-Way Joins

Relaxation

Each query over d tables can be relaxed into d selection queries, similar to Rel-Sel (Section 5.2.4). However, this scheme may suffer from excessive notifications due to the relaxation.

Simple Join Reformulation

On any insertion, we can derive and disseminate all the join tuples produced as a result of the insertion, similar to Ref-J and Ref-J$^+$ in Section 5.2.4. However, the problem of unnecessary data is exacerbated because each insertion into some table would generate all the new joining result tuples (along with the payload for each joining relation in a result tuple). This overhead could be quite large in case many tuples satisfy the join conditions. Furthermore, if there are multiple subscription signatures involving the table of insertion, join results need to be generated separately for every signature. When signatures overlap (e.g., $R \bowtie S \bowtie T_1$ and $R \bowtie S \bowtie T_2$), additional redundancy can arise across results for different signatures.

However, if the join is very selective (i.e., the number of result tuples generated due to an event is small), then this solution may be viable. Using CN* for routing can somewhat mitigate the problem of repeated dissemination of payload.

Fully Extending Two-Way to Multi-Way Joins[2]

We can directly extend the reformulation procedure for binary select-joins in Section 5.3 to consider longer semijoins instead of binary semijoins. We briefly outline the approach below, and point out why it may not work as well as our binary decomposition approach in Section 5.4.

We can group-process all join queries having the same join signature; let $\mathcal{Q}$ denote the corresponding join graph. For each table $R \in \mathcal{Q}$, removing R from $\mathcal{Q}$ would in general result in a set of

[2]Due to its complexity, the reader can skip the details of the extension and directly read the discussion at the end of the section. We do not suggest using this direct extension, but include it for completeness.

(mutually disjoint) connected subgraphs which we call the *remainder graphs* of R; we denote this set by $\bar{\mathcal{R}}$. We define the $R^{\bar{\mathcal{R}}}$-semijoin as a semijoin of the form $R \ltimes (\times_{\mathcal{T} \in \bar{\mathcal{R}}} \bowtie_{T \in \mathcal{T}} T)$, with local selection conditions attached to appropriate tables. Before we outline an approach for handling $R^{\bar{\mathcal{R}}}$-semijoins, we introduce the notion of an *induced projected partial join*:

Definition 5 (Induced Projected Partial Join)**.** *A t_R-induced projected $\mathcal{T}$-partial join, where $\mathcal{T}$ is a connected subgraph of the join graph $\mathcal{Q}$ and t_R is a tuple from a table R connected to $\mathcal{T}$ in $\mathcal{Q}$, is the natural join over $\mathcal{T}$ and $\{t_R\}$, with no selection predicates applied, followed by a projection over the local selection attributes in $\mathcal{T}$. The result tuples of a t_R-induced projected $\mathcal{T}$-partial join can be regarded as points in a $|\mathcal{T}|$-dimensional space called the $\mathcal{T}$-space.*

Insertion into R On the insertion of a tuple t_R into table R, the reformulation for an $R^{\bar{\mathcal{R}}}$-semijoin is a conjunction of predicates, one for each remainder subgraph $\mathcal{T} \in \bar{\mathcal{R}}$. The predicate for each $\mathcal{T}$ is similar to that derived for R-semijoins in the multi-attribute selection case (Section 5.6.3), with the difference that the descriptive skyline (in the $\mathcal{T}^2$-space) in this case is computed for the result tuples of the t_R-induced projected $\mathcal{T}$-partial join.

Insertion into S Here, S can belong to any one of the remainder subgraphs of R. Consider the set of all R tuples that join (directly or indirectly) with the insertion t_S. For each such joining R tuple, say t_R, we generate a message as follows:

- For each remainder subgraph $\mathcal{T}$ not containing S, we include in the message a descriptive skyline for the result tuples of the t_R-induced projected $\mathcal{T}$-partial join (just like the case of inserting t_R described above). Correspondingly, the reformulated subscription includes a condition that ensures that subscription contains at least one point of the descriptive skyline.

- For the remainder subgraph $\mathcal{T}_S$ containing S, consider $\mathcal{J}_{\mathcal{T}_S}$, the set of *old* result tuples of the t_R-induced projected $\mathcal{T}_S$-partial join, prior to the insertion of t_S. We compute the cluster-based descriptive skyline for all *new* result tuples of the t_R-induced projected $\mathcal{T}_S$-partial join (i.e., those involving t_S). Note that points in this cluster-based descriptive skyline are in the $\mathcal{T}_S^2$-space, but by construction of the cluster-based descriptive skyline, they have same

coordinates in each pair of dimensions (left and right endpoints) that correspond to the same dimension in the $\mathcal{T}_S$-space. Therefore, we can "collapse" these points into a set of points P in the $\mathcal{T}_S$-space by removing one dimension from each pair. For each point in P, we compute the skyline envelope of the point with respect to $\mathcal{J}_{\mathcal{T}_S}$. We include both P as well as the skyline envelope points for each point in P in the reformulated message for t_R. Correspondingly, the reformulated subscription includes a condition that ensures that the subscription contains at least one point in P but none of its skyline envelope points.

Finally, the reformulated subscription also checks that t_R satisfies its local selection condition on R.

Discussion With the full-extension approach, the $R^{\bar{\mathcal{R}}}$-semijoins carry all conditions in the original queries, so every tuple in these semijoins participates in the final result of the original queries. In contrast, our binary decomposition approach in Section 5.4 does not offer this guarantee. On the other hand, as with the case of the multi-attribute extension in Section 5.6.3, the higher dimensions pose practical issues. The reformulated messages are larger and the reformulated subscriptions contain more complex (though still stateless) predicates. Processing efficiency also suffers. Although binary semijoins lose some filtering power, they are simple to implement and efficient in practice. Furthermore, breaking queries up into binary semijoins creates more opportunities for group processing and dissemination, while the full-extension approach may end up with many more groups and fewer semijoins per groups.

5.7 Evaluation

5.7.1 Setup, Metrics, and Workload

Server Setup At the server, we implemented all our novel schemes of Enum-SJ, Ref-SJ-Tup, Ref-SJ-Clu, Ref-SJ-Clu$^+$, and Ref-SJ-Sub. For comparison, we also implemented Enum-J, Rel-Sel, Ref-J, and Ref-J$^+$. Refer to Table 5.1 for a summary of solutions. Enum-J uses SSI [AXYY06] for

Method	Delivery	Technique	Comment
Select	CN/CN*	Rel-Sel	Inject events in CN/CN* w/o joining (Sec. 5.2.4)
Select-Join	Unicast	Enum-J	Compute join results for each sub (Sec. 5.2.4)
	CN/CN*	Ref-J, -J+	Inject join results in CN/CN* (Sec.5.2.4)
Select-Semijoin	Unicast	Enum-SJ	Compute semijoins for each sub (Sec. 5.3.1)
	CN/CN*	Ref-SJ	Inject semijoin results in CN/CN* (Sec. 5.3.2); four flavors: -Sub, -Tup, -Clu, -Clu+

Table 5.1: Summary of solutions.

computing select-join results. We support tuple inserts, deletes, and updates. The implementation writes its output to local disk with a write speed of ~70 Mbps, which is roughly similar to a dedicated OC1 optical (~52 Mbps) connection to the Internet. Our data structures use main memory use main memory for optimal performance. The experiments were performed on a set of dual-core Intel Xeon 2.0GHz machines running Linux kernel 2.6.18.

Network Setup We evaluate network performance by implementing a simulator for large-scale networks. The simulator generates application-level routing traces that can be analyzed using our link-level simulator which uses a 20,000 node topology produced by INET [CGJ+02], a generator of Internet-like network topologies. A subset of 1000 nodes act as brokers. In this work, we focus only on measurements between overlay nodes because we have observed that IP-level costs generally follow similar trends as node-level costs.

The vanilla CN we chose to implement is a CAN [RFH+01]-based overlay network that uses a semantic routing space, with subscriptions mapped as points in the space based on their interests.

The semantic space is divided into *zones* based on load balancing criteria, with one broker assigned to each zone. Each zone is responsible for a region of the semantic space, and subscriptions in that region are assigned to the corresponding broker. Zones have knowledge of only their neighbors, and routing proceeds in a multi-hop manner until the region of interest described by the injected message is covered. A similar CN has been used in several other publish/subscribe systems, e.g., [GSAA04, CXY06]. We also implement and report results using our CN* extension. Each CN* node maintains a directory and repository following the LRU replacement policy.

Evaluation Metrics We track both server- and network-side metrics. At the server, we measure the average processing time per event, including both server processing cost and the output of messages to be injected into CN. On the network side, we track: 1) *Network traffic* per event, which measures the total bytes transferred between overlay nodes. 2) *Number of overlay message hops* per event, which measures the total number of messages sent between nodes. 3) *Node stress* per event, which measures load on a node. In this work, we report byte stress, which denotes the total traffic originating from a node. 4) *Hop latency*, which measures the number of overlay hops for an event update to completely reach a subscription. Hop latency roughly corresponds to subscription notification latency, assuming uniform network delays between nodes.

Workload For binary select-joins $R \bowtie S$ (see Example 5), we generate synthetic subscriptions as follows. Let $N(\mu, \sigma)$ represent a normal distribution with mean μ and standard deviation σ. Refer to Table 5.2 for the summary of parameters. Each subscription uses normal distributions distributions N_1 and N_2 to generate the centers of ranges over $R.A$ and $S.C$ respectively. The range centers are located in either low or high portions of event space, to model corresponding user interests. Range widths are derived using normal distributions as well (see Table 5.2).

We experiment with synthetic and real event workloads. The synthetic event workloads use 100 unique values of the join attribute. R tuples are inserted for each unique join attribute value, and 70% of R tuple insertions produce at least one join result.[3] The total number of R tuples in the

[3]This parameter was derived after examining our real stock event workload for the fraction of stocks having at least one rating.

parameter	value
domain of attributes	$[0, 100\text{k}]$
number of subscriptions	100k–1M
$R.A$ range centers	$\mathbf{N}(30\text{k}/70\text{k}, 10\text{k})$
$R.A$ range widths	$\mathbf{N}(20\text{k}, 5\text{k})$
$S.C$ range centers	$\mathbf{N}(15\text{k}/85\text{k}, 6\text{k})$
$S.C$ range widths	$\mathbf{N}(6\text{k}, 5\text{k})$
number of events	44k–74k
number of R tuples in DB	1k–31k
N_1	$\mathbf{N}(25\text{k}/75\text{k}, 3\text{k})$
distribution of $R.A$	$\mathbf{N}(N_1, 8\text{k})$
number of S tuples in DB	100–10100
N_2	$\mathbf{N}(50\text{k}, 10\text{k})$
distribution of $S.C$	$\mathbf{N}(N_2, 3\text{k})$

Table 5.2: Summary of parameters.

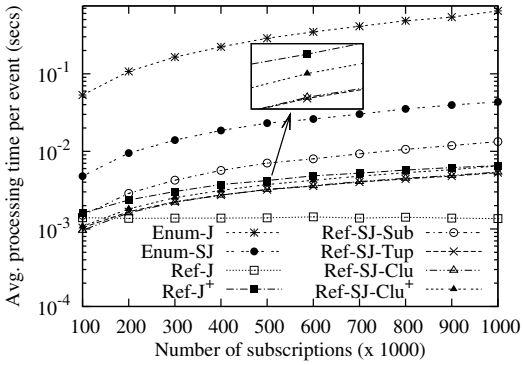


Figure 5.6: Processing time; increasing number of subscriptions.

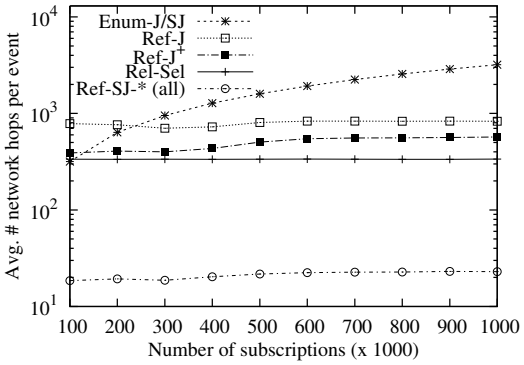


Figure 5.7: Network hops; increasing number of subscriptions.

database is kept constant by deleting older tuples when necessary. The number of S tuples for each join attribute value follows a truncated Zipf distribution with parameter 0.8. We also experiment with a real event workload based on stock data from Yahoo! Finance [Yah] (Section 5.7.2 has for details). Finally, the workloads for our general mix of n-way join subscriptions are described in Section 5.7.4.

Repeatability To verify repeatability across runs, we perform each experiment multiple (up to 10) times, by varying the random seed for the event workload. We found the variation across runs to be minimal—for more than 90% of data points, the 95% confidence interval falls within $\pm 7\%$ of the respective mean. Given the significant difference (often orders of magnitude) across the approaches being compared, we plot only the mean value across runs.

5.7.2 Binary Select-Joins, Unmodified CN

We first examine the benefits of our novel schemes, without considering the added benefits of CN*. We show that even without CN*, our techniques can easily outperform simpler techniques. Unless otherwise indicated, these experiments use 100k subscriptions, with R and S tables having 16k and 5.1k tuples respectively. Each tuple has a payload of 100 bytes.

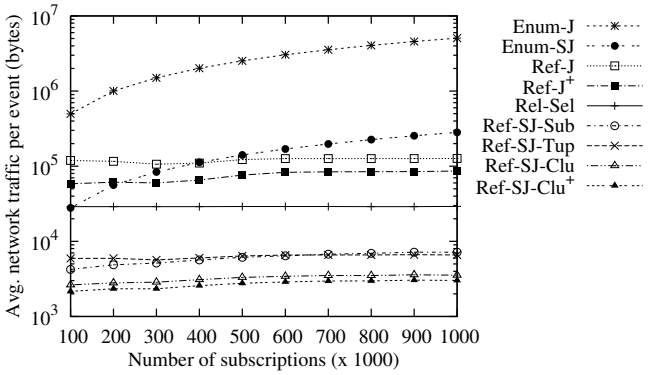


Figure 5.8: Network traffic; increasing number of subscriptions.

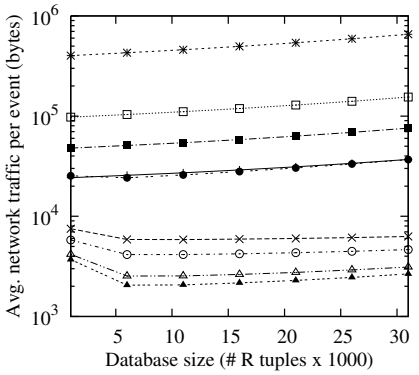


Figure 5.9: Network traffic; increasing database size.

Varying Number of Subscriptions In this set of experiments, we test scalability by varying the number of subscriptions from 100k to 1 million, and measure average costs per event (over 59k events). Note that the y-axis is logarithmic in all the results due to the orders of magnitude differences between various approaches.

The results for server processing time are shown in Figure 5.6. We see that Enum-J is the worst. Even with very efficient processing techniques, the output size dominates and makes this technique perform badly. Enum-SJ is better as it sends minimal data for a particular subscription, but it still suffers from inter-subscription redundancy. The simple reformulations (Ref-J and Ref-J$^+$) perform better, with Ref-J$^+$ being worse as it needs more processing to skip unnecessary join results. Semijoin reformulations (Ref-SJ-Sub, Ref-SJ-Tup, Ref-SJ-Clu, Ref-SJ-Clu$^+$) are very efficient. Ref-SJ-Sub is slower as it has to compute a skyline of affected subscriptions for each event. The compression techniques of Ref-SJ-Clu and Ref-SJ-Clu$^+$ introduce very less overhead over Ref-SJ-Tup.

In terms of overlay message hops (Figure 5.7), our techniques are at least an order of magnitude better. All semijoin reformulation techniques use the same number of hops (they differ only in the size of the skyline). Subscription relaxation (Rel-Sel) does worse due to tuples being unnecessarily sent to brokers. The server-based techniques degrade linearly with increasing number of subscriptions. Ref-J and Ref-J$^+$ also incur a large number of hops.

Network traffic (Figure 5.8 for Enum-J is extremely high as expected. Enum-SJ is much better,

# subs.	100k	300k	500k
Enum-J	484.8	1464.0	2461.5
Enum-SJ	27.3	82.0	137.6
Ref-J$^+$	13.4	14.3	14.6
Rel-Sel	0.86	0.95	0.88
Ref-SJ-Clu$^+$	0.38	0.38	0.39

Table 5.3: Average server stress (kilobytes).

# subs.	100k	300k	500k
Ref-SJ-Sub	73.12	99.54	111.84
Ref-SJ-Tup	312.78	305.88	302.85
Ref-SJ-Clu	29.59	38.15	42.87
Ref-SJ-Clu$^+$	16.88	21.96	24.68

Table 5.4: Average description size (bytes).

but degrades quickly with number of subscriptions. The simple reformulations (Ref-J and Ref-J$^+$) are better due to sharing of costs over CN, but they also incur unnecessary traffic due to result representation and current-content redundancies. Relaxation (Rel-Sel) is slightly better, but at the expense of disseminating and adding irrelevant state at subscriptions. Our semijoin reformulations avoid unnecessary dissemination while sharing costs across subscriptions. Ref-SJ-Clu$^+$ incurs the lowest traffic overall, at least an order of magnitude lower than simpler schemes.

Table 5.3 compares the node stress at the server across various techniques, with Ref-SJ-Clu$^+$ representing the four semijoin reformulation techniques. As expected, Ref-SJ-Clu$^+$ generates the lowest stress, while enumeration-based techniques consume orders of magnitude more outgoing bandwidth at the server. From Table 5.4, we see that the descriptive skylines generated by Ref-SJ-Clu$^+$ are the most compact (with fewest number of points), beating more naive skylines (Ref-SJ-Sub and Ref-SJ-Tup) by a wide margin.

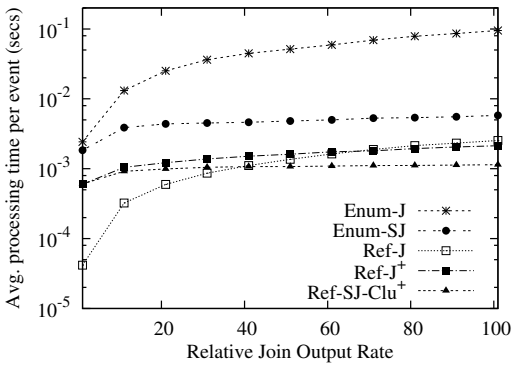


Figure 5.10: Processing time; increasing relative join output rate (RJOR).

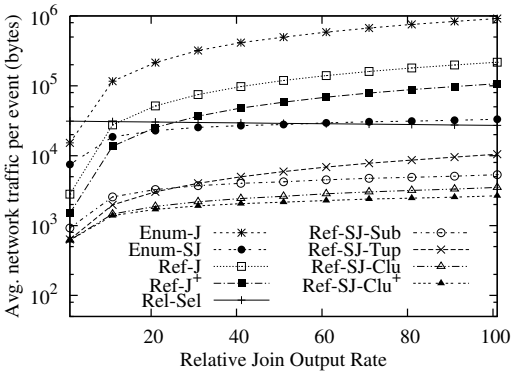


Figure 5.11: Network traffic; increasing RJOR.

Varying Database Size We now see the effect of increasing the number of R tuples in the database (older tuples are deleted when new ones are inserted, to keep the table size constant). Figure 5.9 shows the average network traffic. All schemes have a slightly increasing trend because a larger R table implies fewer deletes, which are cheaper to disseminate. Other factors being equal, a smaller database implies that a new R tuple is likely to cause more subscriptions to need the joining S tuples, because it is less likely that a subscription already has a different R tuple with the same join attribute value. Hence, without CN*, semijoin reformulations degrade slightly in performance at low database sizes. However, we are still able to easily outperform other approaches.

Varying Relative Join Output Rate *Relative join output rate* (*RJOR*) is the average number of join result tuples generated for each inserted event. Like join selectivity, RJOR can impact the performance of some algorithms. Figure 5.10 shows the server processing cost for increasing RJOR. We control RJOR by varying the number of tuples in table S. We see that Ref-J is very good at low RJOR, but quickly degrades due to output size. Ref-SJ-Clu$^+$ scales well with increasing RJOR. Figure 5.11 shows the network traffic for increasing RJOR. Again, semijoin reformulations schemes are clearly superior. The simple Ref-SJ-Tup degrades due to increasing S table size, but the other semijoin reformulations do well even at high RJOR. Although Ref-J and Ref-J$^+$ are good at low RJOR (due to lower result representation redundancy), they quickly degrade with increasing RJOR. Ref-SJ-Clu$^+$ is usually more than an order of magnitude better than the strawman techniques.

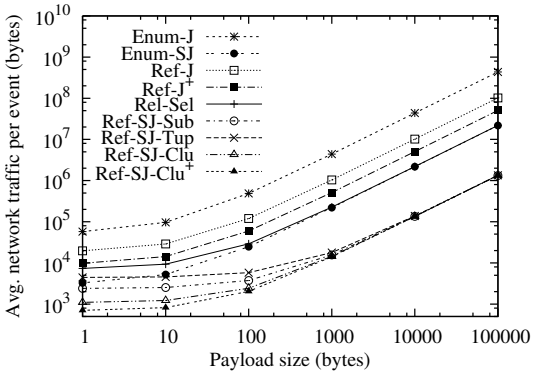


Figure 5.12: Network traffic; increasing payload size.

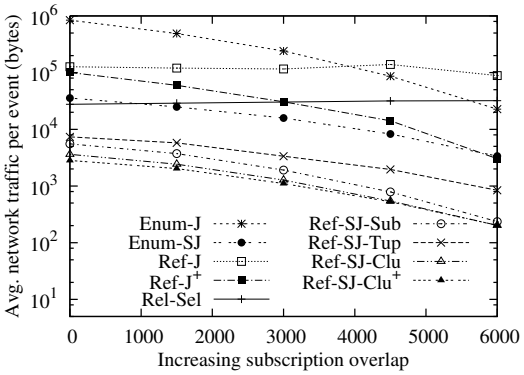


Figure 5.13: Network traffic; increasing subscription overlap.

Increasing Payload Size We increase the payload size (of both R and S tuples) from 1 byte to 100kB, and show the effect on network traffic in Figure 5.12. Note that the x-axis also uses a logarithmic scale. As payload size increases, all approaches incur additional traffic, but the absolute difference in performance in much larger for larger payloads. With increasing payload size, the differences between the various semijoin reformulations diminish because payload size dominates over the description.

Increasing Overlap of Subscriptions We keep the number of subscriptions constant at 100k and increase the amount of overlap of subscriptions by reducing the standard deviation of the distribution from which subscription range centers (for $R.A$ and $S.C$) are drawn. We set the standard deviation of $R.A$ to $13000 - 2x$ and that of $S.C$ to $7500 - x$, where x is varied from 0 to 6000. We plot the performance of various approaches in terms of network traffic, with increasing overlap of subscriptions (increasing x), in Figure 5.13. As we increase the overlap, fewer subscriptions are affected by an update because of the concentration of interests in narrow regions of space. Ref-J is unaffected by overlap since it sends joining tuples regardless of subscriptions. On the other hand, Ref-J$^+$, which takes subscriptions into account, shows lower network traffic as the overlap increases, due to fewer affected subscriptions. Enum-J, Enum-SJ, and the semijoin reformulation schemes also see a reduced traffic with increasing overlap due to the same reason. Among the Ref-SJ approaches, the performance improvement is least for Ref-SJ-Tup since the descriptive sky-

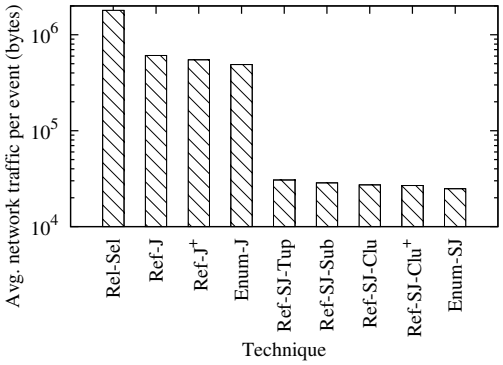


Figure 5.14: Network traffic; considering last hop.

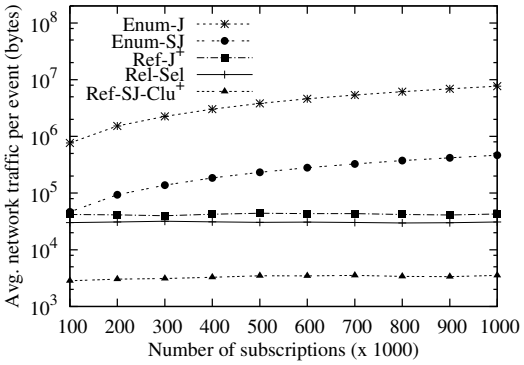


Figure 5.15: Network traffic; real event workload.

line is computed independent of subscription clustering. Ref-SJ-Sub, Ref-SJ-Clu, and Ref-SJ-Clu$^+$ converge in performance at high subscription overlap due to very high amount of clustering. Finally, Rel-Sel actually degrades in performance with increasing subscription overlap because it does not take the join into account, and more overlap (clustering) means that events that fall in the "hot" region of $R.A$ (which many events do) have to be sent to lots of subscriptions.

Considering Last Hop We have ignored the "last hop" from broker to subscriber because we have focused on the performance of the core publish/subscribe middleware, and the final delivery mechanism (e.g., unicast, email, IM, etc.) might be different for different clients. We now examine the effect of considering the last hop, assuming direct unicast from brokers to clients. We assume that the same approach (join, semijoin, or relaxation) is applied until the end subscriber.[4] We see from Figure 5.14 that considering the last hop makes semijoin much more attractive than before. Enum-SJ incurs the lowest total traffic because it avoids the network of brokers completely, at the cost of very high server stress. Ref-SJ-Clu$^+$incurs only slightly higher cost, with the important benefit of sharing dissemination using the broker network.

Results of Real Workload We gather real data from Yahoo! Finance [Yah] to model Example 5. We obtain historical price-to-earning ratios (PER) of 100 random stocks, for a period of 7

[4]Hybrid schemes where semijoin or relaxation is applied inside the broker network, but precise join results are sent to subscribers, are also possible but are omitted for simplicity.

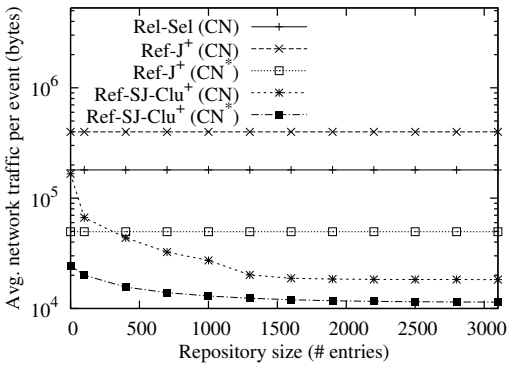


Figure 5.16: Network traffic; increasing repository size.

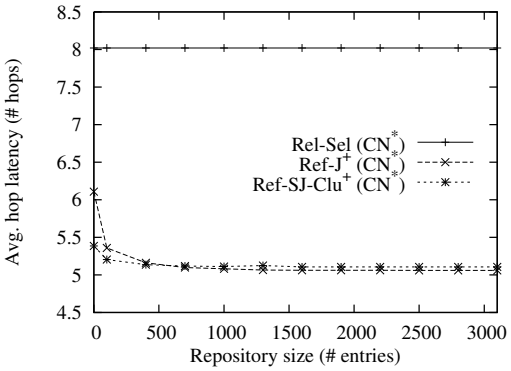


Figure 5.17: Hop latency; increasing repository size.

months. The PER values are mapped to the range $[0, 100\text{k}]$ for use with our subscription workload. Stock ratings (ranging from 1 to 5) are also gathered, mapped, and perturbed using a normal distribution to derive 2300 unique stock ratings from the original set of 460 ratings. Subscription traces are the same as before. Figure 5.15 shows network traffic, as we increase the number of subscriptions. Enum-J and Enum-SJ are very expensive as expected, and degrade with number of subscriptions. Ref-J$^+$ performs better than before because the RJOR is lower (around 23). Still, Ref-SJ-Clu$^+$ is at least an order of magnitude better than the other schemes.

5.7.3 Results of Adding CN*

We now consider the additional benefits derived by using CN* which can reduce re-dissemination redundancy, thus improving performance. We only show Ref-SJ-Clu$^+$, and Ref-J$^+$ (with and without CN*) since these were the best reformulation-based approaches. We also show Rel-Sel for comparison (Enum-J and Enum-SJ do not use CN*).

Effect on Traffic We set the payload directory to be $512kB$, and vary the repository from 0 to 3100 entries. Both R and S tuples carry payloads of 1000 bytes. The R table size is kept very low (500 entries) to expose the worst case performance of Ref-SJ-Clu$^+$. Figure 5.16 shows the network traffic. We see that CN* is able to avoid re-dissemination redundancy for Ref-SJ-Clu$^+$ and Ref-J$^+$, even at low repository sizes. Ref-J$^+$ benefits more since it disseminates more unnecessary

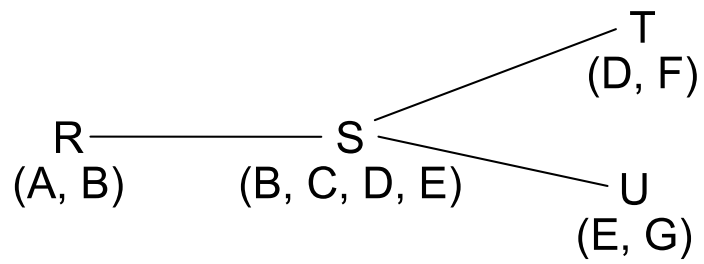


Figure 5.18: Example multi-way join graph.

data, leaving a greater scope for CN* to reduce costs. Ref-SJ-Clu$^+$ is better overall. Note that even with 0 repository size, the directory reduces cost by having only affected brokers pull data from the server. Finally, we found that a repository of just 400 entries can reduce server stress by 7 times for Ref-J$^+$ and 3 times for Ref-SJ-Clu$^+$ compared to using CN.

Effect on Hop Latency Figure 5.17 shows the average hop latency across all subscriptions and events. R table size is very low (200 entries) to further penalize Ref-SJ-Clu$^+$. We see that even at low repository sizes, the potential extra roundtrip does not increase the hop latency by much. Again, Ref-SJ-Clu$^+$ is impacted minimally at low directory sizes because it relies less on CN* to perform well. Note also that Rel-Sel has a much higher hop latency, since a message needs to reach many more subscribers dispersed over a larger number of brokers.

5.7.4 Multi-Way Select-Joins

We use the join graph in Figure 5.18, and experiment with a mix of 50k $R \bowtie S$, 50k $R \bowtie S \bowtie T$, 50k $R \bowtie S \bowtie T \bowtie U$, and 20k $S \bowtie T \bowtie U$ queries (all with selection predicates). The subscriptions and events are derived using normal distributions similar to the case of binary joins. We experiment with two event schema.

General Schema Here, R, S, T, and U have 10, 70, 70, and 30 tuples per unique join attribute value, respectively. Enum-J and Ref-J were found to be prohibitively expensive due to the large number of join results for the multi-way joins. Rel-Sel was found to generate 23kB traffic per event, around 9 times worse than the optimal Ref-SJ-Clu$^+$ semijoin decomposition. In Figure 5.19, we

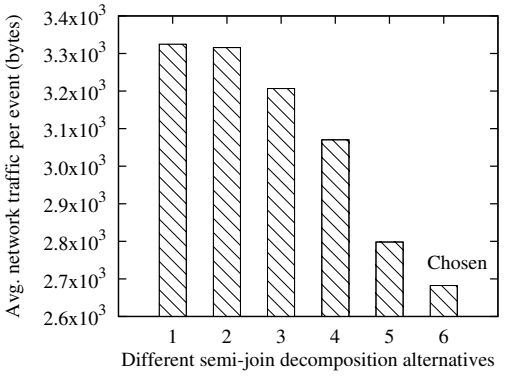


Figure 5.19: Network traffic; multi-way join mix.

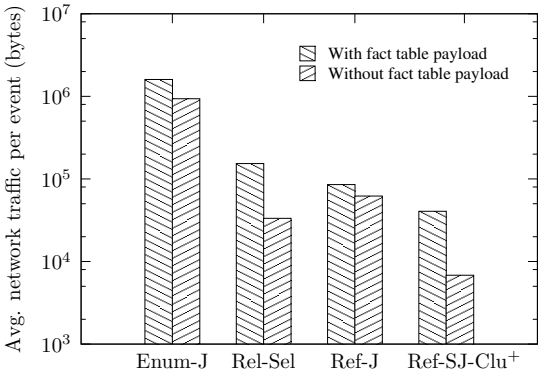


Figure 5.20: Network traffic; star schema mix.

compare the costs of different semi-join decompositions (without CN*).Our cost model orders the semijoin group costs as: $c(S^R) > c(S^U) > c(S^T)$. The greedy assignment of query groups is optimal in this case, and gives the lowest traffic. Some other random assignments are also shown. The optimal decomposition is around 20% better than the worst decomposition.

Star Schema We experiment with the popular star schema. Here, there are 3 dimension tables (R, T, and U) with one fact table (S). The query graph is the same as before. We model 1000 entries in each dimension table, and 25000 entries in the fact table. The dimension tables each have a payload of 100 bytes. The star schema is an extreme case where each insertion into the fact table produces exactly one join result. Thus, RJOR is very low, and Ref-J can do well. Figure 5.20 shows the results. When the fact table has no payload, Ref-SJ-Clu$^+$ is 5, 9, and 127 times better than Rel-Sel, Ref-J, and Enum-J respectively, because it disseminates the bulky dimension table tuples only when necessary. Ref-J has to send out complete join results. The advantage is lesser when the fact table has equal payload (100 bytes) because it diminishes the relative advantage (all schemes have to send the bulky S tuples for each S insertion). Nevertheless, we find that Ref-SJ-Clu$^+$ outperforms other techniques (even without CN*).

5.8 Related Work

Continuous Query Systems Continuous query systems (e.g., [LPT99, CDTW00, DGH+06]) can be regarded as a form of publish/subscribe, where continuous queries over streams correspond to our subscriptions. NiagaraCQ [CDTW00] supports select-join processing at a server. CACQ [MSHR02] group-processes filters, and supports dynamic reordering of joins and filters. PSoup [CF03b] exploits set-oriented processing on joins with arbitrary join conditions. These systems correspond to Enum-J: They ignore the dissemination aspect and do not jointly optimize processing and dissemination. Consequently, they cannot avoid the redundancies intrinsic to producing traditional join results. Cayuga [HDG+07] supports queries joining two XML streams, but their schemes also ignore dissemination and are optimized for value joins over XML. We focus on relational select-joins, support multi-way joins, and consider both processing and dissemination.

Publish/Subscribe Systems Several publish/subscribe systems have made the subscription language more powerful (e.g., [DRF04, JS03, CXY06, CPY07, FJLM05].). *SMILE* [JS03] supports SQL queries, while *PADRES* [FJLM05] supports subscriptions that can express correlations across events. These systems add application-specific logic and state into the network and do not optimize for group-processing or disseminating select-join subscriptions with varying selection predicates. They operate similarly to Ref-J, and can reduce only inter-subscription redundancy. We process queries efficiently, reduce all types of redundancies, and use a simple CN interface for efficient dissemination. Our techniques can be employed by these systems to handle a large number of multi-way select-join queries efficiently. In earlier work [CXY06, CPY07], we have used reformulation to support complex queries over CN. However, [CXY06] focuses on range aggregation, while [CPY07] tackles subscriptions with value-based notification conditions.

Distributed Joins Distributed join processing systems, where state is distributed across overlay nodes, correspond to Rel-Sel if selects are applied first. PIER [HHL+03] supports SQL queries (including joins) over DHTs, but targets one-time queries and does not optimize for multiple subscriptions. Idreos et al. [ITK06] support two-way joins over overlay networks by re-indexing queries and

routing tuples to them. This can incur high overhead because each query may be replicated for every unique join attribute value, and selects are done only as post-processing. Ahmad et al. [ACJZ05] tackle distributed joins, but they focus on network locality and data locality issues, with the objective of reducing delay. These systems add complexity by designing new distributed schemes with application-specific logic and state in the network. We optimize processing and dissemination of a mix of multi-way select-join queries, and use the simple, stateless, off-the-shelf CN interface, making our novel techniques easy to deploy and manage, yet ensuring very high efficiency.

Other Related Work Semijoins have been employed by many systems [BGW+81, SKBK01] to reduce communication in distributed databases. Work on view maintenance (e.g., [LR05, Huy00, QGMW96]) also considers joins. However, they do not address the problem of simultaneously supporting a large number of select-joins. Moreover, like Enum-SJ, they do not reduce redundancies across queries and updates.

5.9 Conclusions

A publish/subscribe system needs to optimize both subscription processing and result dissemination, particularly for complex queries such as joins. In this chapter, we aimed to develop an end-to-end solution to support a large mix of multi-way select-join subscriptions. Towards this goal, we identified several key redundancies in traditional techniques of supporting such subscriptions. Our novel semijoin-based reformulation schemes reduce these redundancies and outperform standard techniques by orders of magnitude. The schemes are easy to deploy and maintain, yet ensure very high efficiency. We also proposed a new extension (CN*) to content-driven networks, that further reduces redundancy of disseminating bulky payloads. Extensive experiments on real and synthetic workloads with two-way and multi-way select-joins validated the benefit of our schemes, and showed orders of magnitude improvement compared to standard techniques, for both server and network metrics.

Chapter 6

ProSem: Scalable Wide-Area Publish/Subscribe

We saw in Chapter 1 that there has been a lot of work on publish/subscribe systems in both the database community (on efficient subscription processing) and the networking community (on notification dissemination). However, in this dissertation, we showed that fundamentally, a wide-area publish/subscribe system must efficiently handle subscription processing as well as notification dissemination (henceforth called *prosemination* to emphasize their inseparability). In earlier chapters, we developed holistic prosemination techniques for many types of subscriptions. These techniques offer orders-of-magnitude improvement in performance over traditional, non-holistic schemes. Based on these findings, we have built a publish/subscribe system called *ProSem*, named after "prosemination" to signify our novel holistic approach to prosemination. Besides joint optimization, ProSem has several additional features that also distinguish it from other systems:

- ProSem supports complex, stateful subscriptions, e.g.: "keep me informed of stocks with the minimum price-to-earning ratio in a certain risk range," "notify me when the price of IBM has changed by more than 3%," etc. Furthermore, ProSem aims at supporting such subscriptions on an Internet-scale. Beyond exploiting common subscription predicates, ProSem uses subscription indexing techniques specially developed for certain subscription types to scale to millions of subscriptions.

- Using novel reformulation techniques, ProSem is able to support complex, stateful subscriptions on top of standard network substrates that otherwise only support stateless subscriptions. This approach avoids pushing complicated application logic into the network, enforces a clean interface between the server and the network, and allows the use of an "off-the-shelf" content-driven network (CN)—a term that was introduced in Chapter 2 and encompasses many network substrates including content-based networks [CRW01] and overlay networks supporting range searches.

- ProSem recognizes that the best prosemination strategy may differ for different events and different groups of subscriptions. Therefore, ProSem uses cost-based optimization to choose the best prosemination strategy among a set of available alternatives at run-time, according to user-defined system-level objectives.

ProSem is our first attempt at combining and implementing the building blocks introduced in earlier chapters, across various subscription types in one system. ProSem's run-time optimization of prosemination is also original.

6.1 Preliminaries

As introduced in Chapter 2, we model a publish/subscribe system as consisting of (1) publishers who publish *events* which are modeled as updates to a database, (2) subscribers who declare their interests in data (called subscriptions) as queries over the database, and (3) the middleware—a server and an optional network of brokers—responsible for supporting the subscriptions. As a running example, consider the database view $\texttt{Stock}(\texttt{Symbol}, \texttt{Price}, \texttt{Risk}, \texttt{PER})$ tracking the up-to-date information for each stock, including the stock symbol, current price, risk factor, and price-to-earning ratio, a popular measure of stock quality. For simplicity, suppose that published events follow the schema $\langle \texttt{Symbol}, \texttt{Price}, \texttt{Risk}, \texttt{PER} \rangle$, and the attributes reflect their new values after the update.

Recall from Chapters 1 and 2 that a subscription is *stateless* if we can determine whether and how to update it by examining an incoming event itself, without referring to the database, event history, or per-subscription state. For instance, a subscription of the form "`SELECT * FROM Stock WHERE` $x_1 \leq$ `Risk AND Risk` $\leq x_2$" is stateless; it is affected by any event with $\texttt{Risk} \in [x_1, x_2]$. On the other hand, a subscription is *stateful* if we cannot process it by simply examining the incoming event. Consider "`SELECT MIN(PER) FROM Stock WHERE` $x_1 \leq$ `Risk AND Risk` $\leq x_2$." This subscription is stateful because it may or may not be affected by an incoming event, depending on whether there is some other stock with $\texttt{Risk} \in [x_1, x_2]$ and `PER` lower than that of the event. Moreover, if an event raises the `PER` of a stock, subscriptions that previously had it as their

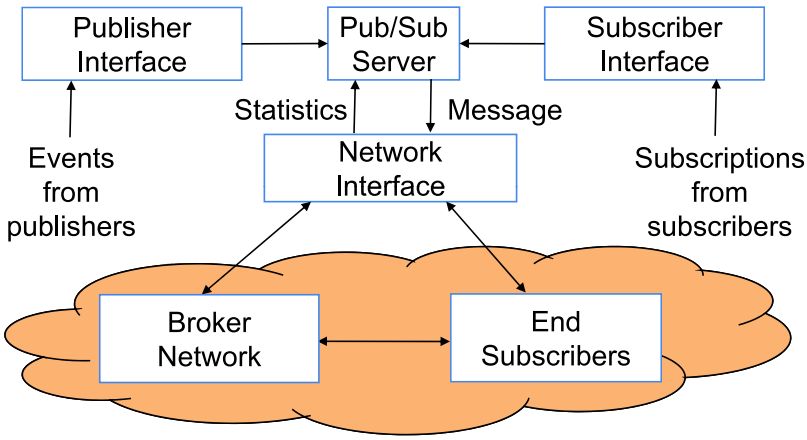


Figure 6.1: Overall design of ProSem.

minimum may need to look for new answers in their respective risk ranges.

6.2 System Design

6.2.1 Overall Architecture

ProSem processes subscriptions by grouping them into *subscription types*. A subscription type corresponds to a basic subscription template, with parameters that can take on values to create a subscription instance of that type. For example, all range-min subscriptions with template "`SELECT MIN(PER) FROM Stock WHERE` $x_1 \leq$ `Risk AND Risk` $\leq x_2$" have the same type, where each instance is parametrized by (x_1, x_2). Dividing subscriptions into groups allows us to apply specialized indexing and group-prosemination techniques for each type. Figure 6.1 shows the overall system design of ProSem. In the remainder of this section, we describe the main components of the overall architecture in detail. In the next subsection, we focus on the ProSem server.

Publisher Interface

Publishers can directly deliver events to the server using a TCP socket connection, through a publisher interface. As an alternative, we can use webpage parsers that interface with standard HTTP web servers and poll Web pages or RSS feeds to generate events. We can define multiple webpage

parsers, one for each publisher website. A webpage parser periodically loads the publisher webpage using the HTTP client, parses it and generates events that are sent to the publisher interface. The publisher interface reads the incoming event and adds it to an event buffer at the central server. We can also plug in a synthetic event generator to experiment with synthetically created events to test the system.

Subscription Interface

New subscriptions are registered with the ProSem server through a subscriber interface, using socket communication. An end subscriber registers a subscription by specifying the subscription type and parameters. The details regarding all supported subscription types are maintained by the server in an XML file. The client gets these details by retrieving the XML file when it is started up. Each subscription is assigned a globally unique *subscription ID* by the server, and this subscription ID is returned to the end subscriber by the subscriber interface.

Network Interface

The ProSem server processes events and uses the network interface to send notification messages to subscribers, either with direct IP unicast or through an overlay network of brokers in CN. ProSem relies on standard, off-the-shelf CN network substrates that do not by themselves support application state; instead ProSem uses reformulation (Chapter 2) to support stateful subscriptions over a stateless network interface. The network interface can interact with either a real deployment of CN or a simulator.

Subscriber Client

In order to receive updates and notify users, we have developed a subscriber client. The subscriber client runs on the user's machine, and is responsible for creating/deleting subscriptions as well as receiving notification messages. The client supports multiple concurrent subscriptions by the same user. Figure 6.2 shows a screenshot of the client.

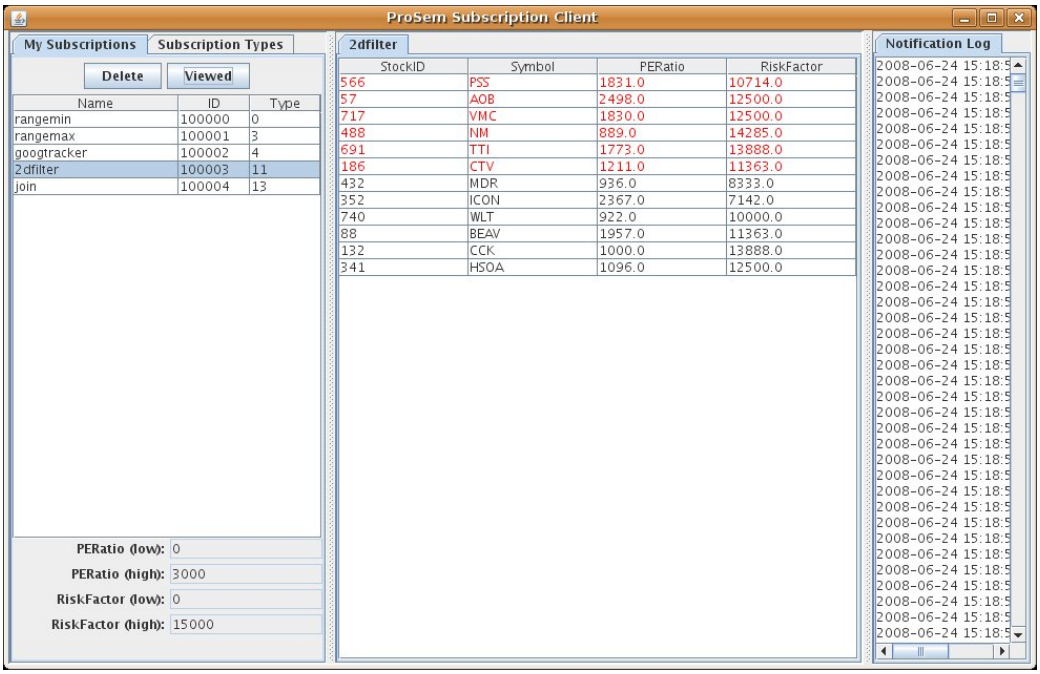


Figure 6.2: The ProSem subscriber client.

When the subscriber client is started, it connects to the server and retrieves the set of supported subscription types, as well as the update rules and transformation rules (see Section 2.4) associated with each subscription type. The rules are implemented as SQL queries which operate on a user-level instance of SQLite [SQL]. The client allows users to create and delete subscriptions, using the "Subscription Types" tab shown in Figure 6.2. When a subscription is added by the user, the database tables specific to that subscription instance are generated, and the subscription request is sent to ProSem's subscriber interface. Subscription deletions are handled in a similar manner. Whenever a notification message is received from ProSem by the client, the client executes the appropriate update rules to update the local database state, and then refreshes the user view based on the transformation rule (query), as shown in Figure 6.2 (center). A log of received updates is displayed on the right. Updates to the subscription view after the last time that the result was "viewed" by the user as marked in red color.

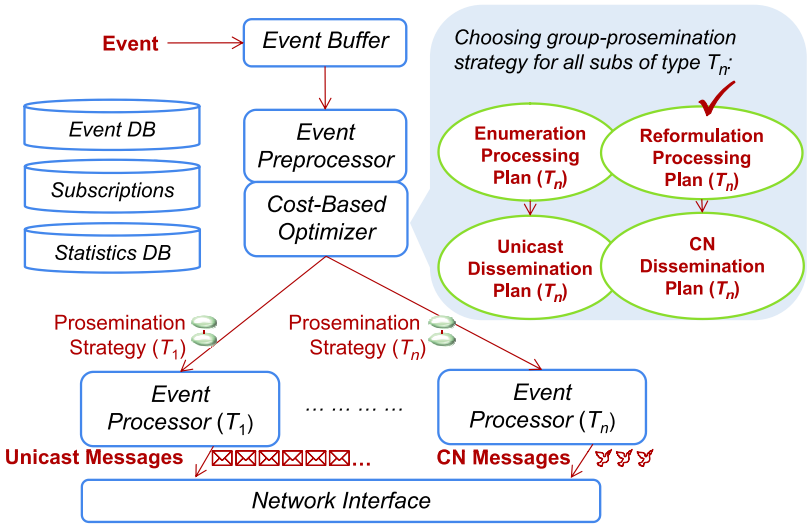


Figure 6.3: Server architecture.

6.2.2 Server Architecture

Figure 6.3 shows the architecture of the ProSem server. The server maintains the database (of events and related information) as well as the set of active subscriptions. We describe the main server components below.

Subscription Processor

The ProSem server processes a new subscription by reformulating it (see Chapter 2) and indexing the reformulated subscriptions in CN. In addition, the server also indexes the subscription locally (grouped by type). The initial content of the subscription is computed at the server and sent to the subscriber directly as a notification message.

Event Preprocessor

Recall that events are written by the publisher interface to an event buffer. This ensures that if the event rate is temporarily higher than processing rate, no events are discarded. The event preprocessor reads one event at a time from the event buffer, updates the affected database table, and looks up the metadata to find out which subscription types are potentially affected by the event.

There is a specialized event processor for each subscription type. An optimizer sends the event to each affected specialized event processor, to determine the optimal processing and dissemination strategy (some possible strategies are discussed in Section 6.3) for that event. Next, the optimizer chooses an optimal ordering of subscription types and processes the event in that order, using the specialized event processors described next.

Event Processors

A specialized event processor is responsible for processing an event with respect to a particular subscription type. The event is first routed to the event optimizer. The job of this component is to first enumerate all possible choices of both network dissemination technique and associated server-side processing technique. It uses statistics to choose the optimal execution plan for the event, for the particular subscription type. For example, if the chosen strategy is reformulation using certain data structures and CN instances, the processor would perform the appropriate processing and generate zero or more reformulated messages as output. The message (or messages) generated by the processor are sent to the network interface for dissemination. We discuss ProSem's cost-based optimization in detail in Section 6.4.

6.3 Prosemination Strategies

In Section 6.2, we saw that at a high level, ProSem operates as follows:

- ProSem reads each incoming event from an event buffer and sends the event to the event preprocessor.
- The event preprocessor determines (by analyzing schema and subscription templates) the set of subscription types potentially affected by the event.
- ProSem then uses a *cost-based optimizer* to choose the best prosemination strategy for each subscription type.

- The chosen strategies are then carried out by the *event processors* (specialized for each subscription type).

- After the event has been processed, the server updates the database and associated indexes accordingly, and moves on to the next event.

In this section, we present details on various prosemination strategies. In Section 6.4 we discuss how to choose among them using the run-time cost-based optimizer.

6.3.1 Stateless Subscriptions

ProSem supports two alternatives for stateless subscriptions. First, the server can compute the affected subscriptions and unicast to each one. Second, it can use a content-driven network (CN) deployed over a set of brokers, where each broker is responsible for notifying a subset of the subscriptions. CN directly supports stateless subscriptions: It allows message recipients to declare their interests to the network as predicates over message contents; senders simply inject messages into CN, which automatically forwards them to all recipients with matching interests.

6.3.2 Stateful Subscriptions

For stateful subscriptions, ProSem supports several alternatives which vary in the way they interface server processing and network dissemination.

- *Enumeration*. The server computes the list of affected subscriptions along with notification contents for each of them. Notifications are then sent out by unicast. This approach is similar to the first alternative above for stateless subscriptions; nevertheless, handling a large number of stateful subscriptions at the server requires novel indexing and processing techniques.

- *Relaxation*. We relax stateful subscriptions into stateless ones to be handled by CN. Then, we rely on post processing at the brokers to compute updates to the original stateful subscriptions. For e.g., a range-min subscription can be made stateless by dropping `MIN`. The broker effectively maintains a range subscription, from which updates to range-min can be

derived. The downside is that maintaining the "larger" range subscriptions at each broker leads to much higher notification traffic from the server.

- *Reformulation*. Figure 2.7 illustrates reformulation. For each incoming event, the server dynamically reformulates it into messages embedded with state information, and then injects them into the network. Stateful subscriptions can then be reformulated into stateless ones over the reformulated messages, since the additional embedded state now allows these subscriptions to be processed using the message contents alone. The stateless reformulated subscriptions can be handled directly by a standard CN substrate.

While the strategies above are general, detailed techniques differ across different types of subscriptions. Reformulation in particular requires unconventional indexing and processing techniques, because it changes the role of processing from computing a list of affected subscriptions to reformulating messages, which are essentially *semantic descriptions* of affected subscriptions. ProSem uses reformulation techniques that (as we showed in earlier chapters) can outperform more conventional strategies for a variety of subscriptions.

Note that we have deliberately chosen a simpler system design for ProSem than other stateful systems that push processing of application state into the network, e.g., *SMILE* [JS03]. All prosemination strategies above feature clean server/network interfaces and rely on stateless off-the-shelf network components. We believe this design makes ProSem easier to deploy and maintain on a large scale.

Another important disadvantage of systems that use stateful brokers is that the use of stateful brokers ties the system to a specific choice of dissemination, because all relevant state updates have to go through the brokers in order to maintain accurate state at the brokers. This means that such systems cannot switch to a different technique (such as enumeration) for certain events. Our approach does not have this problem—we will see next in Section 6.4 that this flexibility allows us to use a powerful dynamic cost-based optimizer to improve system performance.

Publish/subscribe system	Database system
Event	Query
Dissemination plan	Execution plan
Overlay maintenance	Index update
Event history statistics	DB statistics (e.g., histograms)

Table 6.1: Parallels between a publish/subscribe system and a DBMS.

6.4 Cost-Based Optimization

6.4.1 Introduction

Depending on the event and subscription characteristics, it may happen that different techniques need to be employed by the publish/subscribe system for optimal performance. The choice of dissemination plan (e.g., broadcast, multicast, unicast, CN, etc.) is sensitive to the workload. Similarly, at the server, the use of different processing plans (e.g., data structures and algorithms) may be appropriate under different scenarios. This means that we need an optimization framework that chooses the appropriate *prosemination plan* for each event at runtime.

Optimization should be guided by both online statistics incorporating both events and subscriptions. From a DBMS perspective, this is similar to treating different server processing and network dissemination techniques as available indexes (with the associated costs) whose usage is driven by the query optimizer. An important component of the optimizer is a cost model that defines the cost of a particular prosemination plan. Table 6.1 illustrates the high-level parallels between query optimization in a database system, and technique selection in a wide-area publish/subscribe system.

The available choices for processing and dissemination are usually based on subscription type characteristics. For example, if most subscriptions of a particular type require every published event (for example, in case of a `SELECT *` subscription type), the best strategy may be to use broadcast (with no database or subscription state at the server) and rely on filtering at subscribers to eliminate false positives, if any. In case of stateless subscription types (e.g., stateless filters), simple

server matching with unicast or directly interfacing with CN are good options. In case of stateful subscription types where most subscriptions are rarely affected by incoming events (e.g., range-MIN), we may maintain more state (with index structures) at a server and use either traditional continuous query processing with enumeration (unicast) or reformulation with CN to reach the affected subscribers.

It is generally beneficial to take per-event statistics into account while making optimizer decisions. For example, if an event is expected to affect very few subscriptions, we may prefer to use unicast as the dissemination mechanism. For an incoming event, we need to use available statistics, assign a cost to each available prosemination plan based on the cost-model, and finally choose the best prosemination plan.

6.4.2 The ProSem Cost-Based Optimizer

Overview

ProSem draws parallels from query optimization in a traditional DBMS: alternative prosemination strategies are similar to alternative query execution plans in a DBMS. The available choices differ across subscription types, and depend on the available server indexes and network components. For each subscription type, the best choice may vary based on the incoming event being processed, runtime statistics, and system performance goals. To provide flexibility and adaptivity, ProSem uses a cost-based optimizer to select a good prosemination plan for each incoming event at runtime. A plan specifies a prosemination strategy for each subscription type, and the order in which to execute them. We next describe how to cost each plans and find the best plan.

Cost Model

ProSem models the system cost under four basic categories of metrics.

- *Server Time* is measured as the inverse of throughput, which is the number of events per second that ProSem can support during continuous operation. It accounts for both *server*

processing time (SPT) for computing outgoing notification messages and *server dissemination time (SDT)* for putting these messages on the network interfaces.

- *Response Time* is the delay from an event arrival until an affected subscription receives its notification. The average response time accounts for SPT, SDT, average propagation delay through the network, as well as the processing order of subscription types (those processed later experience longer response times).

- *Server Stress* measures the total outgoing traffic from the server.

- *Bandwidth Consumption* measures the total traffic incurred by ProSem between pairs of communicating nodes in the overlay network (middleware) for notification dissemination. The last hop from brokers to subscribers is uniform across all approaches and can be ignored.

We define the *system cost* as a weighted combination of the four metrics above. Our goal is to choose a prosemination plan that minimizes system cost. Both the objective and constraints of the optimization depend on the application (e.g., target response times) and the market (e.g., server cost versus bandwidth cost). ProSem allows system administrators to make dynamic adjustments to the objective weights.

Metrics Estimation

To estimate the cost of a prosemination plan, ProSem continuously monitors statistics relevant to the four basic metrics. Another critical piece of information needed to optimize for each incoming event is the number of affected subscriptions for each subscription type. Recall that reformulation in effect computes semantic descriptions of affected subscriptions. Using these descriptions, ProSem can estimate the number of affected subscriptions without enumerating them—just like a DBMS estimates query result size—by maintaining summary statistics about subscriptions.

Each event processor maintains a data structure for metrics estimation. We use *ANN* [ANN], a data structure that supports very fast approximate nearest neighbor search queries on data points of high dimensionality. Recall that our semantic descriptions of affected subscriptions are shapes

(regions) in high-dimensional space. We index these descriptions as points in ANN, and store with each description the statistics associated with that description.

The statistics collected include SPT, SDT, network traffic, average notification latency, server stress (outgoing traffic from the server), and the number of affected subscriptions. These statistics are collected from the network and server components which process events. We rebuild the ANN index periodically using the collected statistics, imposing a user-defined limit on the size (memory usage) of each data structure. On an incoming event, the associated semantic descriptions are first computed using our reformulation techniques. We lookup each semantic description in ANN, to determine the associated statistics for the closest matching descriptions. The metrics for the original description are estimated using the statistics for the closest matching description.

Optimizer Strategy

ProSem uses a simple yet effective optimizer based on a greedy strategy that optimizes each subscription type separately. The optimizer enumerates the available alternatives for the prosemination plan, costs each of them based on the cost model, and chooses the strategy that offers the lowest estimated weighted cost. The weights assigned to the four metrics are knobs specified by the user. This optimization strategy works well because the space of available plans is much smaller than, for example, in traditional databases. With increasing subscription types and strategies supported, we will need a more complex optimizer, and this is an interesting area of future work. When deciding the processing order among subscription types, ProSem orders the subscription types in increasing value of per-subscription server costs, in order to reduce the overall average response time.

6.5 Evaluation

6.5.1 Setup, Metrics, and Workload

Setup We tested our implementation of ProSem on a laptop with a 2.0GHz Intel Centrino Duo processor and 3GB RAM. The laptop runs both the server and a network simulator that we use as

the network interface. We define 14 subscription types supported by the system, including various forms of range-aggregation, value-based notification subscriptions, select-joins, and stateless filter subscriptions over one or more attributes. The system can easily be expanded to add support for more subscription types.

Metrics We measure various performance metrics of the system, including:

- Server throughput, in events per second.
- Percentage of reformulation, the percentage of decisions taken by the optimizer that used reformulation.
- Latency, the average delay from event processing to notification reaching a subscriber.
- Server stress, in terms of outgoing traffic from server per event.
- Middleware traffic, in terms of bytes transferred between overlay nodes per event (this excludes the last hop to the client).

Workload We model a stocks application, with three tables in the system. A table `Stocks` contains information on stock symbols, price-to-earning ratios, and risk factors. A second table `Reviews` has information on stock reviews and user ratings. Finally, a separate table `StockPrices` tracks the exact price updates to some stocks. We mine real stock data from Yahoo! Finance and feed it to our system as updates to the `Stocks` and `StockPrices` tables. Updates to the `Reviews` table are synthetically generated. Our workload has a total of around 800k events (some of our collected data are replayed multiple times to generate sufficient traces for our experiments).

We use a synthetic subscription workload of 100k subscriptions. Of these, 30k are range-aggregation subscriptions (on the `Stocks` table), 30k are subscriptions with value-based notification conditions (on the `StockPrices` table), 5k are one-dimensional filters, and 5k are two-dimensional filters (on the `Stocks` table).

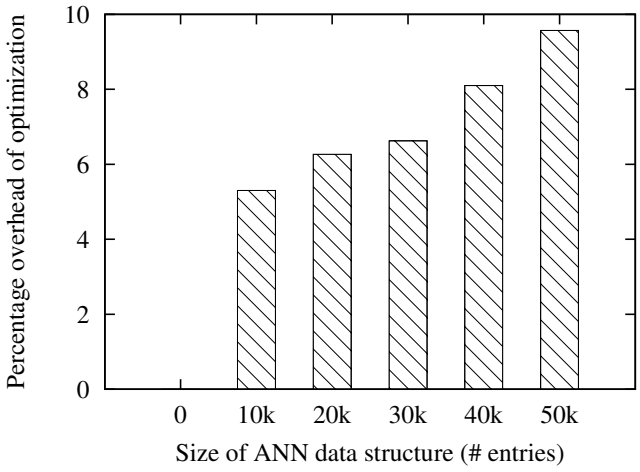


Figure 6.4: Overhead of cost-based optimization.

6.5.2 Performance Results

Overhead of Cost-Based Optimization

The main cost incurred during per-event cost-based optimization is the lookup of the ANN data structure (we found the cost of searching the space of plans to be negligible). To measure the impact of this cost, we increase the size of the ANN data structure from 0 to 50k entries (per subscription type), and plot the percentage overhead incurred over the base case where the optimizer is bypassed completely. To ignore the effects of the variation in the optimizer's decisions across data points, we force the system to use a reformulation-based prosemination plan regardless of the optimizer's decision. Note that this choice exposes the worst-case overhead of optimization because the reformulation-based plan has the lowest processing time (we found the percentage overhead in a mixed plan using default knob weights to be around an order of magnitude lower). Figure 6.4 shows the result of this experiment. We see that the overhead incurred by our cost-based optimizer is very low, and is less than 10% even in the case of 50k entries per subscription type in the ANN data structures. In fact, as we will see next, an ANN data structure of just 1000 entries is sufficient to give near-optimal decisions in the case of our workload.

Accuracy of Cost-Based Optimization

In order to examine the accuracy of our dynamic cost-based optimizer, we plot the weighted system cost per event (averaged over a moving window of 200 events) over a fixed event range, for various optimizer strategies. We use the default knob settings for this experiment. To compare against our cost-based optimizer, we experiment with three alternatives: *Random* (where plans are chosen randomly), *Worst* (where the plan with the highest estimated cost is always chosen), and *Oracle* (where the optimal plan with the lowest cost assuming perfect knowledge is always chosen). We experiment with four versions of our cost-based optimizer, which use 1, 10, 100, and 1000 entries per ANN data structure respectively.

The results are shown in Figure 6.5. As expected, *Worst* shows highest cost and is followed by *Random*, while *Oracle* has the lowest cost. As we increase the size allocated to ANN, the performance of our cost-based optimizer improves dramatically. With a size of 1000 entries, the optimizer is nearly as good as *Oracle*. The reason for the good performance even at low ANN sizes is that the differences in costs across different prosemination plans are usually quite large. As a result, low to moderate variations in the estimated cost often do not impact what prosemination plan is finally chosen by the optimizer.

Impact of *Server Time* Knob

We examine the impact of giving greater importance to server processing time by increasing the weight assigned to the *Server Time* knob of ProSem. We increase the knob from 0 to 10k. Figure 6.6 shows the results. As we increase the knob value, ProSem tends to favor reformulation as it has lower server processing time. Thus, both server throughput (Figure 6.6 upper left) and percentage of reformulation (Figure 6.6 upper right) increase as seen in the figures. We also observe that server stress has reduced as reformulation has lower server stress, but at the expense of increased traffic in the middleware (see Figure 6.6 lower center).

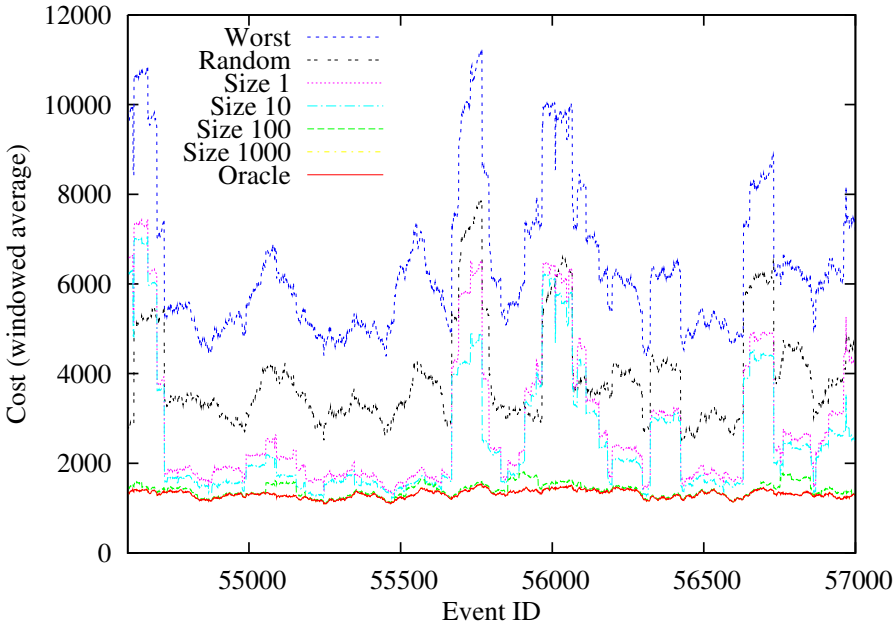
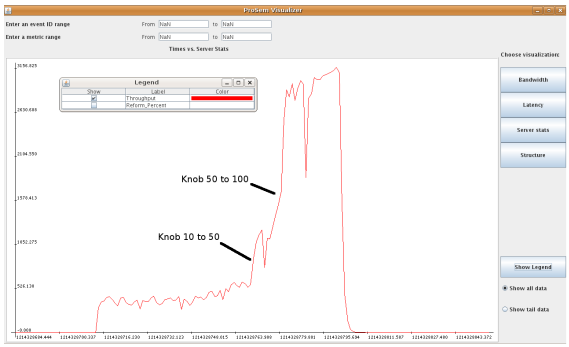


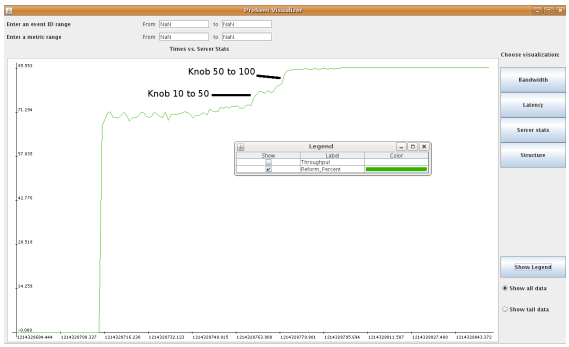
Figure 6.5: Accuracy of cost-based optimization.

Impact of *Response Time* Knob

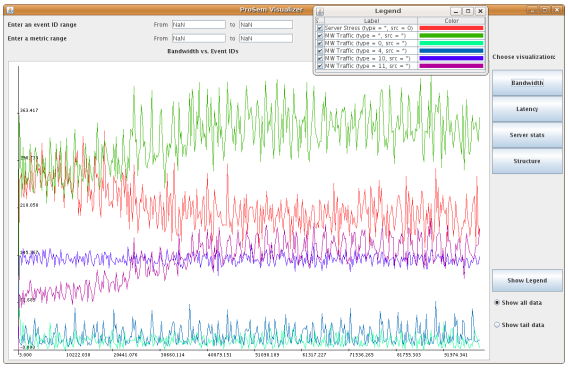
We examine the impact of giving greater importance to the delay before subscribers receive notifications, by increasing the weight assigned to the *Response Time* knob of ProSem. We increase the knob from 0 to 50 in this experiment. Figure 6.7 shows the results. As we increase the knob value, ProSem tends to favor unicast (although it has higher server processing time) for smaller number of affected subscriptions, as it provides quicker update delivery as compared to using a network of brokers. Thus, both server throughput (Figure 6.7 left) and percentage of reformulation (not shown) decrease. We also observe (Figure 6.7 right) that latency reduces with increasing knob value. The reduction is not marked because we show the average notification latency, and the system continues to choose reformulation for events that affect a large number of subscribers, because reformulation remains the best alternative for such events (unicast degrades too much when notifying many subscriptions).



Server throughput vs. time.

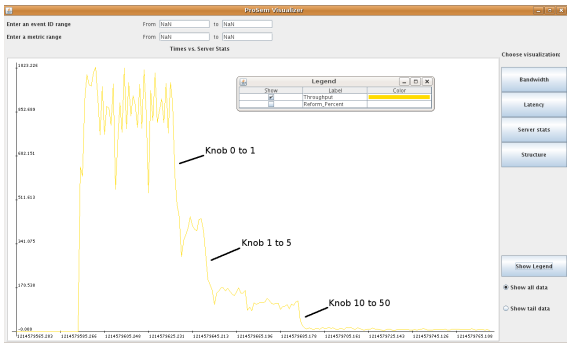


Percent reformulation vs. time.

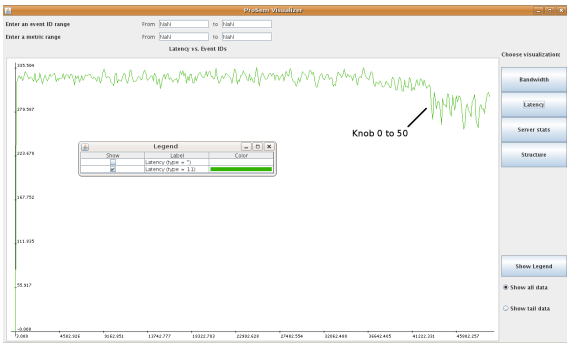


Middleware traffic vs. event ID.

Figure 6.6: Impact of increasing *Server Time* knob.



Server throughput vs. time.



Notification latency vs. time.

Figure 6.7: Impact of increasing *Response Time* knob.

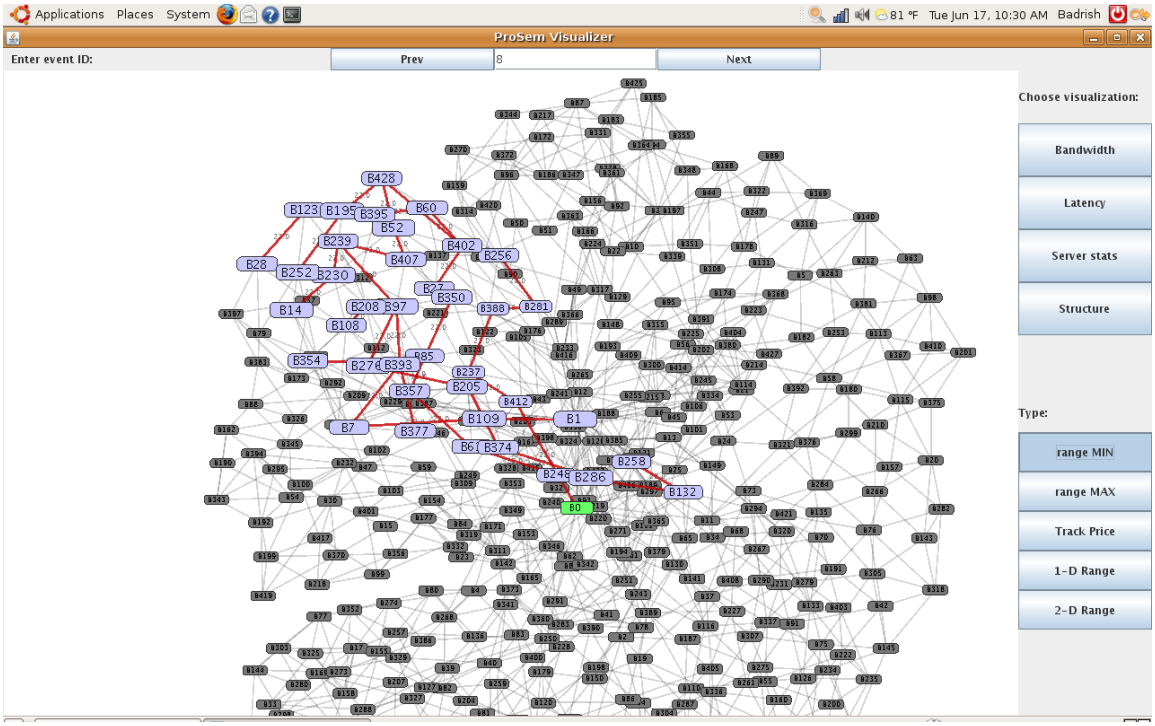


Figure 6.8: Visualizing network dissemination.

Impact of Other Knobs

As we increase the *Bandwidth Consumption* knob, we see an effect similar to the effect of increasing the knob for latency. Specifically, increasing the knob drops the total middleware traffic, at the expense of increasing server stress due to greater usage of unicast. The difference between this knob and the knob for latency is that increasing this knob always favors unicast (which incurs no middleware traffic), whereas increasing the latency knob favors the strategy that offers lower latency (not necessarily unicast in case many subscriptions are affected). Finally, increasing the *Server Stress* knob tends to favor reformulation, resulting in lower stress at the expense of higher middleware traffic.

Visualizing Network Dissemination

We have built a visualizer to demonstrate the overlay routing performed by ProSem's CN interface. Figure 6.8 shows the visualization for an instance of ProSem using 429 brokers (one of which is the

server, shown in green). The brokers use a CAN-based overlay network (see Section 2.2) topology. The end subscribers are not shown in the visualization. Figure 6.8 also demonstrates the dissemination of an update to the affected subset of a total of 30k range-min subscriptions (the network path taken by the notification message is highlighted in the figure). Since reformulation reaches precisely the affected subscribers, and an update is expected to affect the answer to relatively fewer range-min subscriptions, we see that this update reaches only a small percentage of brokers.

6.6 Conclusions

In this chapter, we described the design, implementation, and evaluation of ProSem, our publish/subscribe system based on the ideas and techniques introduced in this dissertation. ProSem's cost-based optimizer was shown to provide very high accuracy at low overhead. The knobs exposed by ProSem are effective in controlling various aspects of system performance. A natural next step is to deploy ProSem over a live network such as *PlanetLab* [Pla], which would allow normal Internet users to use our client to register subscriptions and receive notifications in real time.

Chapter 7

Future Work and Conclusions

In this chapter, we first discuss several interesting ideas for future work in the research areas covered by this dissertation. We finally provide some concluding remarks and observations.

Areas of Future Research

Open Challenges in Stateful Publish/Subscribe

With the advent of Web 2.0, we are witnessing a proliferation of dynamic user-generated content, leading to the challenge of *large-scale publishers* (complementing the challenge of large-scale subscribers that my dissertation addresses). The data needs themselves are getting even more complex, with requests such as "*deliver to my desktop, the ten most popular related blog entries published within two hours of a calamity*". As we move to an Internet-scale continuous query processing model, we need to build systems that can cope with the new challenges: (1) How can we efficiently match the large number of publishers of dynamic data, to the expressive subscribers? (2) How can we support generalized SQL-style queries with hierarchies of complex query subplans (e.g., aggregation over join results)?

One strategy to support generalized queries is to again adapt concepts from traditional databases, by treating queries as data and events as *meta-queries* over this "data" to return affected queries. How far can we push this analogy — does relational algebra suffice to express such meta-queries to support arbitrary data needs? The advantage of such a view is twofold: (1) It provides a structure to the problem, making it more tractable. (2) It gives us a well-defined framework for optimizing support for complex data needs.

Batched Event Processing and Dissemination

An important area of future work is the examination of batched event processing and dissemination. Intuitively, it is useful to batch incoming events as long as the consistency requirements are met. Batching is beneficial because processing a batch of updates is typically more efficient than processing the events individually. Further, dissemination of a batch of events opens up the potential for optimized group dissemination as compared to the dissemination of each event separately. Work in this area will involve examining the tradeoffs involved in batching, and developing robust techniques that take advantage of situations where batching can give us improvement in system cost while meeting the user-defined fidelity or performance goals.

Overlay Network Design for Publish/Subscribe

An oftentimes overlooked but important area of research in content-based publish/subscribe systems is the problem of designing networks that are suitable for wide-area publish subscribe. Recall that a traditional content-based publish/subscribe system consists of an overlay network of brokers that cooperate to disseminate data. A data source registers itself with some broker, which then acts as the source of data. An interested subscriber registers its interest with some broker. A content-based routing tree is rooted at the source broker and spans all the interested brokers, and events are thus sent to the brokers which need the data. The broker themselves send the data to all subscribers which are registered with them and are affected by the event, using an appropriate last hop delivery mechanism such as unicast, email, instant messaging, etc.

How can we assign subscriptions to brokers, to optimize for the dual constraints of network and data locality? Prior work has assumed that subscribers either attach themselves to the closest broker [ASS$^+$99], or use simple semantic considerations (like the simple exclusive partitions [DRF04]) to decide which broker to attach to. Neither alternative is attractive: the former may cause every event to be delivered to nearly every broker, while the latter may require many subscribers to attach themselves to brokers that are located very far away.

To clarify the problem, assume that we have a content-based publish/subscribe system with attributes from a set A, in the event schema. Further, let there be k brokers and N subscribers. The

event space is a $|A|$-dimensional space over all possible events, where any event can be mapped into a point in the event space. A subscription provides values (or ranges) for some non-empty subset A' of the attributes, and can be mapped to a shape in the event space. For example, if $|A| = 2$, a subscription can be a point, horizontal or vertical line, or a rectangle in the two-dimensional event space. If we assume that subscriptions are all range-subscriptions, every subscription would be a rectangle in the event space. We assume knowledge of a window of the E most recent events.

The general assignment problem consists of three phases:

(1) **Partitioning**: Create a *partitioning* P of the event space, which consists of k partitions of the event space. The partitions need not be disjoint, but need to cover every possible subscription. For any partitioning P, every event in the event history stabs one or more partitions. That event needs to reach all the stabbed partitions. We define the filtering power [DRF04] of a partitioning to be the average number of partitions stabbed by an event from the event history. Thus, we have an infinite number of possible partitionings, each with a potentially different filtering power. In general, a partition can be any shape, and it does not have to be a single contiguous chunk. However, as a simplification, we can impose some constraint on the shape. For example, we may require that partitions be rectangular or a union of two rectangles, and so on.

(2) **Broker assignment**: Given a partitioning P_i, we have to assign the k partitions in this partitioning to the k brokers (one partition per broker). There are $k!$ possible broker assignments for every partitioning,

(3) **Subscriber assignment**: Given a partitioning and an associated broker assignment, the next step is to assign subscriptions to brokers. A subset of partitions whose union covers a subscription is called a covering subset of the subscription. A subscription could have many possible covering subsets (the entire set of partitions is trivially one such covering subset). We can focus on only the *minimal* covering subsets (there could be more than one unique minimal covering subset). Minimal covering subsets are simply those covering subsets that do not contain a subset which is itself a covering subset. For each minimal covering subset (for a particular partitioning and broker assignment), we define a per-subscriber assignment cost which is a function of the latencies from the subscriber to all the brokers that are assigned the members of that covering subset. The subscriber

assignment cost is the total cost over all subscribers.

For any given partitioning, one can identify the lowest cost subscriber assignment for each of the $k!$ broker assignments. The overall cost of the partitioning and broker assignment, then, is a function of the filtering power and the lowest-cost subscriber assignment for that partitioning and broker assignment. The optimal solution is the partitioning, broker assignment, and subscriber assignment that yields the lowest overall cost.

The general solution to this problem may be intractable, and there is a need to develop heuristics that do well in practice. The heuristics would involve the use of collected statistics. For example, the broker assignment and subscriber assignment phases use statistics on latencies between subscribers and all the brokers in the system. This could be approximated using techniques such as Global Network Positioning to map nodes to points in a network space. Event statistics may need to be maintained, potentially as a histogram over the event window. These statistics would help in the partitioning phase. For instance, if there are a large number of events in a certain region of the event space, we may want only one partition to cover that region (assuming that the partition has the resources to handle the event rate), so that these events do not have to be sent to more brokers. The goal is to reduce the filtering power of a partitioning, and this is computed based on the available dynamic event statistics.

Concluding Remarks

The number of data sources (e.g., RSS feeds, stock updates, network monitoring logs, etc.) as well as data demands by end-users has increased tremendously through the last decade. With more than a billion Internet users in an increasingly networked world, data management has become a critical factor in large distributed applications. A publish/subscribe system provides an ideal middleware to match data source to data consumers.

This dissertation showed how we can push the capability of publish/subscribe systems, and enable scalable support for complex and expressive database-style subscription models in a large-scale wide-area publish/subscribe system. We showed that by performing smart data management

with a novel holistic approach incorporating both database processing and network dissemination, we can reduce costs and create powerful systems to support such subscriptions. Such systems are more efficient, use less network resources, and are ready to cope with the large scales and volumes of the data and queries of tomorrow.

As part of our effort to achieve the above goal, we closely examined a spectrum of possible solutions based on the interface between the database (publish/subscribe server) and the dissemination network, in Chapter 2. We showed that the various points on this spectrum (unicast, multicast, CN, CN$^+$, S-CN, and DS-CN) each have their advantages and disadvantages along various subjective (e.g., implementation and management cost) and objective (e.g., bandwidth and server stress) metrics. Based on this study of the spectrum of interfaces, we proposed message and subscription reformulation (whose details were provided in Chapter 2) as a general technique to support complex subscriptions using the simple and well-interface of a content-driven network. The advantage of reformulation is that it allows one to use an off-the-shelf stateful CN as the dissemination network to support stateful subscriptions. In Chapter 2, we also showed that reformulation techniques can be classified into a spectrum based on the general approach used to perform the reformulation.

We showed that the geometric mapping approach to reformulation, while less general because of the need to tailor to individual subscription types, can provide orders-of-magnitude higher efficiency in both event reformulation cost (using novel algorithms) and network traffic (using CN), for a large class of common subscription types. Accordingly, in Chapters 3, 4, and 5, we showed how reformulation can be applied to efficiently group-process and group-disseminate a broad range of subscription types including range-aggregation (MIN, MAX, SUM, COUNT, AVG), range-DISTINCT, multi-way select-joins, and subscriptions with value-based notification conditions. In each case, we made insightful findings into the characteristics of the subscription type and leveraged those findings to achieve orders-of-magnitude improvement in performance (in terms of both server processing time and network traffic).

Finally, in Chapter 6 we showed how our research ideas were incorporated to build a flexible high-performance publish/subscribe system named ProSem, which supports multiple subscription types and performs run-time cost optimization to choose the appropriate processing and dissemi-

nation strategies for each incoming event, for each supported subscription type. The middleware administrator can tune the system using knobs to specify weights attached to various costs (processing time, notification latency, middleware bandwidth, and server stress).

There are some common underlying design philosophies that have been incorporated in this dissertation. The first philosophy is to use simple, generic, yet powerful interfaces to glue complex subsystems together. Adhering to simple interfaces requires additional work—more careful analysis/design and a keener understanding of the problem—in order to hide the complexity from the interface. However, it reaps dividends by enabling easier deployment, reuse, and adoption with oftentimes much better performance than specialized interactions. The second philosophy is to take advantage of joint group-processing and group-dissemination. Group-processing at the server requires the design of new data structures and algorithms that exploit the flexibility of generating *descriptions* of destinations (affected subscriptions) rather than enumerating them. Group-dissemination works in conjunction with the server using reformulation, and leverages an efficient and well-studied large-scale dissemination abstraction (content-driven networks).

This dissertation unifies databases and publish/subscribe systems at two levels. First, at the level of functionality, we add scalable support for stateful subscriptions that resemble database views. Second, in terms of system architecture, we unify databases and publish/subscribe by incorporating database-style statistics-based query optimization into our publish/subscribe system. In doing so, we are able to exploit the parallels between publish/subscribe events and database queries, and publish/subscribe dissemination structures and database access paths. We show that this unification leads to a powerful, stateful, efficient, yet easily manageable publish/subscribe infrastructure.

Bibliography

[AAFZ95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *SIGMOD*, 1995.

[AASY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, 1997.

[ABB+02] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *2002 ACM Symp. on Principles of Database Systems*, pages 221–232, Madison, Wisconsin, USA, June 2002.

[ABKM01] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, pages 131–145. ACM Press, 2001.

[ACC+03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *SIGMOD*, 2003.

[ACJZ05] Y. Ahmad, U. Cetintemel, J. Jannotti, and A. Zgolinski. Locality aware networked join evaluation. In *NetDB*, 2005.

[AF00] M. Altmel and M.J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 2000 International Conference on Very Large Data Bases*, 2000.

[AFZ] S. Acharya, M. Franklin, and S. Zdonik. Dissemination-based data delivery using broadcast disks.

[Agu84] L. Aguilar. Datagram routing for internet multicasting. In *SIGCOMM*, 1984.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *2000 SIGMOD*, pages 261–272, Dallas, Texas, USA, May 2000.

[AJB98] M. O. Akinde, O. G. Jensen, and M. H. Böhlen. Minimizing detail data in data warehouses. In *1998 EDBT*, pages 293–307, 1998.

[AKSJ03] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *2003 ACM SIGCOMM Internet Measurement Conference*, Miami, Florida, USA, October 2003.

[ANN] ANN: A Library for Approximate Nearest Neighbor Searching. `http://www.cs.umd.edu/~mount/ANN/`.

[ASS+99] M. K. Aguilera, R. Strom, D. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *1999 ACM Symp. on Principles of Distributed Computing*, Atlanta, Georgia, USA, May 1999.

[AV96] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *IEEE Symp. on Foundations of Computer Science*, 1996.

[AXYY06] P. K. Agarwal, J. Xie, J. Yang, and H. Yu. Scalable continuous query processing by tracking hotspots. In *VLDB*, 2006.

[BBK02] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM*, pages 205–217, 2002.

[BCL89] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3):369–400, September 1989.

[BCM+99] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish- subscribe systems. In *ICDCS*, 1999.

[BGW+81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and Jr. J. B. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM TODS*, 6(4):602–625, 1981.

[Bil04] A. S. Bilgin. Incremental read-aheads. In *Proceedings of the ICDE/EDBT Ph.D. Workshop*, March 2004.

[BKS01] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.

[BO03] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, 2003.

[BS05] A. Bulut and A. K. Singh. A unified framework for monitoring data streams in real time. In *ICDE*, 2005.

[BW95] E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. In *Second Internatinal Workshop on Rules in Database Systems*, pages 292–308, Athens, Greece, September 1995.

[CBGM03] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Trans. on Knowledge and Data Engineering*, 15(1), 2003.

[CCRK04] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical internet coordinates for distance estimation. In *International Conference on Distributed Systems*, 2004.

[CDKR02] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communication*, 2002.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[CF02] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[CF03a] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, 2003.

[CF03b] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB J.*, 2003.

[CFF+02] C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of the 2002 International Conference on Very Large Data Bases*, 2002.

[CG00] R. Chirkova and M. R. Genesereth. Linearly bounded reformulations of conjunctive databases. In *DOOD*, 2000.

[cGAA04] O. Şahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.

[CGJ+02] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, 2002.

[CGL+96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *1996 SIGMOD*, pages 469–480, Montreal, Canada, June 1996.

[CH02] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[Chi02] R. Chirkova. The view-selection problem has an exponential-time lower bound for conjunctive queries and views. In *ACM Symp. on Principles of Database Systems*, pages 159–168, 2002.

[CHS02] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. *The VLDB Journal*, 11(3):216–237, 2002.

[Chv79] V. Chvátal. A greedy heuristic for the set covering problem. *Math. Oper. Res.*, 4:233–235, 1979.

[CKL+97] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *1997 SIGMOD*, pages 405–416, Tucson, Arizona, USA, May 1997.

[CLGS04] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *WebDB*, pages 25–30, 2004.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 1977 ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[CNF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[CPY07] B. Chandramouli, J. M. Phillips, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems. In *VLDB*, 2007.

[CRR+05] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *SIGCOMM*, 2005.

[CRW00] A. Carzaniga, D. S. Rosenblum, , and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *2000 ACM Symp. on Principles of Distributed Computing*, 2000.

[CRW01] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 2001.

[CW01] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, 2001.

[CXY05] B. Chandramouli, J. Xie, and J. Yang. On the Database/Network Interface in Large-Scale Publish/Subscribe Systems. Technical report, Department of Computer Science, Duke University, November 2005. `http://www.cs.duke.edu/dbgroup/papers/2006-cxy-pubsubinter.pdf`.

[CXY06] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD*, 2006.

[CY08] B. Chandramouli and J. Yang. End-to-end support for joins in large-scale publish/subscribe systems. In *VLDB*, 2008.

[CYA+08] B. Chandramouli, J. Yang, P. K. Agarwal, A. Yu, and Y. Zheng. ProSem: Scalable wide-area publish/subscribe. In *SIGMOD*, 2008.

[CYV04] B. Chandramouli, J. Yang, and A. Vahdat. Distributed network querying with bounded approximate caching. Technical report, Department of Computer Science, Duke University, June 2004. `http://www.cs.duke.edu/dbgroup/papers/2004-cyv-netcache.pdf`.

[CYV06] B. Chandramouli, J. Yang, and A. Vahdat. Distributed network querying with bounded approximate caching. In *DASFAA*, 2006.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[DCKM04] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[DEB03] Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 2003.

[DF03] Y. Diao and M. Franklin. Query processing for high-volume XML message brokering. In *VLDB*, pages 261–272, 2003.

[DFJ+96] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[DGH+06] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.

[DGM+04] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[DGR03] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.

[Die82] P. F. Dietz. Maintaining order in a linked list. In *ACM Symp. on Theory of Computing*, 1982.

[dMLR07] C. du Mouza, W. Litwin, and P. Rigaux. Sd-rtree: A scalable distributed rtree. In *ICDE*, pages 296–305. IEEE, 2007.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of nonequijoin algorithms. In *1991 VLDB*, pages 443–452, Barcelona, Spain, September 1991.

[DRF04] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.

[DS87] P. F. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *ACM Symp. on Theory of Computing*, 1987.

[ESV03] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM*, 2003.

[FJL+01] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/ subscribe. In *SIGMOD*, 2001.

[FJLM05] E. Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *FIW*, 2005.

[FJP+99] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An architecture for a global internet host distance estimation service. In *IEEE INFOCOM 1999*, 1999.

[FK99] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[FKS03] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.

[Fre] FreePastry. `http://freepastry.rice.edu/`.

[FZ97] M. Franklin and S. Zdonik. A framework for scalable dissemination-based systems. In *OOPSLA*, 1997.

[GGB+97] A. Geissbuhler, J.F. Grande, R.A. Bates, R.A. Miller, and W.W. Stead. Design of a general clinical notification system based on the publish-subscribe paradigm. In *Proceedings of the 1997 American Medical Informatics Association Annual Fall Symposium*, 1997.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *1996 EDBT*, pages 140–144, March 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *1995 SIGMOD*, pages 328–339, May 1995.

[GM99a] A. Gupta and I. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[GM99b] A. Gupta and I.S. Mumick, editors. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999.

[GM99c] Ashish Gupta and Inderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, June 1999.

[GMLY98] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *1998 VLDB*, New York City, New York, USA, August 1998.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *1993 SIGMOD*, pages 157–166, May 1993.

[GO95] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry: Theory & Applications*, 5:165–185, 1995.

[GSAA04] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Middleware*, 2004.

[HCH+99] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE*, 1999.

[HDG+07] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.

[HHL+03] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.

[HJ91] E. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *1991 Workshop on Algorithms and Data Structures*, pages 153–164, 1991.

[Hot] S. M. Hotz. Routing information organization to support scalable interdomain routing with heterogeneous path requirements.

[Huy97] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *1997 VLDB*, pages 26–35, Athens, Greece, 1997.

[Huy00] N. Huyn. Speeding up view maintenance using cheap filters at the warehouse. In *ICDE*, 2000.

[HXYY05] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *2005 ICDE*, Tokyo, Japan, April 2005.

[HY04] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *2004 ICDE*, Boston, Massachusetts, USA, March 2004.

[ITK06] S. Idreos, C. Tryfonopoulos, and M. Koubarakis. Distributed evaluation of continuous equi-join queries over large structured overlay networks. In *ICDE*, 2006.

[JD02] M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *SIGCOMM*, 2002.

[JMR91] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental implementation model for relational databases with transaction time. *IEEE Trans. on Knowledge and Data Engineering*, 3(4):461–473, 1991.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *1995 ACM Symp. on Principles of Database Systems*, pages 113–124, San Jose, California, USA, May 1995.

[JOV05] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

[JS03] Y. Jin and R. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *DEBS*, 2003.

[KCLB04] G. Kollios, J. Considine, F. Li, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

[KK04] R. Kumar and J. Kaur. Efficient beacon placement for network tomography. In *2005 ACM SIGCOMM Internet Measurement Conference*, Sicily, Italy, October 2004.

[Kos00] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[Kot05] Y. Kotidis. Snapshot queries: Towards data-centric sensor networks. In *ICDE*, 2005.

[KRAV03] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[KRE03] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR DB system. In *ICDE*, 2003.

[LJD+00] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, 2000.

[LLL+06] H. Lim, J. Lee, M. Lee, K. Whang, and I. Song. Continuous query processing in data streams using duality of data and queries. In *SIGMOD*, pages 313–324, 2006.

[LM04] I. Lazaridis and S. Mehrotra. Approximate selection queries over imprecise data. In *ICDE*, 2004.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[LR05] B. Liu and E. Rundensteiner. Cost-driven general join view maintenance over distributed data sources. In *ICDE*, 2005.

[LYC+00] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 461–472, 2000.

[MÖ1] G. Mühl. Generic constraints for content-based publish/subscribe. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, London, UK, 2001.

[MCC04] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 2004.

[ML86] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. *SIGMOD Record*, 26(2), 1997.

[MSHR02] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[MWA+03] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[MYY05] K. Munagala, J. Yang, and H. Yu. Online view maintenance under a response-time constraint. In *ESA*, 2005.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. *SIGMOD Record*, 30(2):437–448, 2001.

[NDK+02] S. Nath, A. Deshpande, Y. Ke, P. P. Gibbons, B. Karp, and S. Seshan. Irisnet: An architecture for compute-intensive wide-area sensor network services. In *Intel Research Pittsburgh Technical Report IRP-TR-02-10*, December 2002.

[NZ02] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.

[NZ04] T. S. E. Ng and H. Zhang. A network positioning system for the internet. In *USENIX Annual Technical Conference 2004*, Boston, MA, June 2004.

[OAA+00] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[OJW03] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[Ols03] C. Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.

[OLW01] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, 2001.

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. *ACM Operating Systems Review*, 27(5):58–68, 1993.

[OW02] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, 2002.

[PB02] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[PBMW98] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[PC05] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

[PFJ+01] J. Pereira, F. Fabret, H.-A. Jacobsen, F. Llirbat, and D Shasha. Webfilter: A high-throughput XML-based publish and subscribe system. In *VLDB*, 2001.

[Pla] PlanetLab. `http://www.planet-lab.org/`.

[Pos] PostgreSQL. `http://www.postgresql.org/`.

[Pow96] D. Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996.

[PSCP02] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *2002 VLDB*, Hong Kong, China, August 2002.

[PTFS05] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 2005.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *1996 ACM Workshop on Materialized Views: Techniques and Applications*, pages 110–118, June 1996.

[RCK+95] N. Roussopoulos, C.-M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS project: View R Us. *IEEE Data Engineering Bulletin*, 18(2):19–28, 1995.

[RD01a] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *2001 Middleware*, November 2001.

[RD01b] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.

[RHKS01] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *2001 Intl. Workshop on Networked Group Communication*, 2001.

[RK04] S. Rewaskar and J. Kaur. Testing the scalability limits of overlay routing infrastructures. In *Fifth Passive and Active Measurements Workshop*, Juan-les-Pins, France, April 2004.

[RKK+04] A. Rodriguez, C. Killian, D. Kostić, S. Bhat, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, March 2004.

[Rou91] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM TODS*, 16(3):535–563, September 1991.

[RS02] M. Rabinovich and O. Spatschek. *Web caching and replication*. Addison-Wesley, 2002.

[SDBL07] R. Strom, C. Dorai, G. Buttner, and Y. Li. SMILE: distributed middleware for event stream processing. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 553–554, 2007.

[SDR03] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *VLDB*, 2003.

[SHYY05] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In *ICDE*, 2005.

[SK03] A. Shriram and J. Kaur. Identifying bottleneck links using distributed end-to-end available bandwidth measurements. In *First ISMA Bandwidth Estimation Workshop*, San Diego, California, USA, December 2003.

[SKBK01] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. Integrating semi-join-reducers into state-of-the-art query processors. In *ICDE*, 2001.

[SMK+01] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[SQL] SQLite. `http://www.sqlite.org/`.

[SRR05] S. Shah, K. Ramamritham, and C. V. Ravishankar. Client assignment in content dissemination networks for dynamic data. In *VLDB*, 2005.

[SRS02] S. Shah, K. Ramamritham, and P. J. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *VLDB*, 2002.

[SSK97] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[Sun] Sun Microsystems. Java Message Service. `http://java.sun.com/products/jms/`.

[TA04] P. Triantafillou and I. Aekaterinidis. Content-based publish/subscribe systems over structured P2P networks. In *DEBS*, 2004.

[TAJ03] D. Tam, R. Azimi, and H. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Intl. Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 2003.

[TBF+03] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 1992.

[Traa] Transaction Processing Council. TPC-H. `http://www.tpc.org/tpch/`.

[Trab] Transaction Processing Council. TPC-R. `http://www.tpc.org/tpcr/`.

[VBV03] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 2003.

[VYW+02] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *2002 OSDI*, December 2002.

[WDR06] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

[WVS+99] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP*, 1999.

[Yah] Yahoo! Finance. `http://finance.yahoo.com/`.

[YD04] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, 2004.

[YGM99] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM TODS*, 1999.

[YHSY04] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenance of XML structural indexes. In *2004 SIGMOD*, Paris, France, June 2004.

[YHY+03] K. Yi, Y. Hai, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[YSG03] A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *2003 CIDR*, Asilomar, California, USA, January 2003.

[YV01] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

[YW98] J. Yang and J. Widom. Maintaining temporal views over non-temporal information sources for data warehousing. In *Proceedings of the 1998 International Conference on Extending Database Technology*, pages 389–403, 1998.

[YW00] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment. In *2000 EDBT*, pages 395–412, Konstanz, Germany, March 2000.

[YW01] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *2001 ICDE*, Heidelberg, Germany, April 2001.

[YYY+03] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *2003 ICDE*, Bangalore, India, March 2003.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *1995 SIGMOD*, pages 316–327, May 1995.

[ZKJ01] Y. B. Zhao, J. Kubiatowitcz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, Department of Computer Science, Univ. of California at Berkley, 2001. Tech. Rep. UCB/CSD-01-1141.

[ZZJ+01] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide- area data dissemination. In *2001 Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.

Biography

Badrish Chandramouli is a Ph.D. student in the Department of Computer Science at Duke University. He earned his M.S. in Computer Science from Duke University in 2004. Before Duke, he earned a Bachelor's degree in Computer Science from VJTI, Mumbai University, India. He has worked for two years at the Texas Instruments Research and Development Center in Bangalore, India. He has also interned twice at the IBM T.J. Watson Research Center in New York, USA. He is interested in databases and distributed systems, with a focus on data management, continuous query processing, and wide-area publish/subscribe systems. He is also interested in query processing challenges in modern database management systems. He has also worked on enabling pervasive access to business data and applications.

As of July 2008, his publications include:

- Badrish Chandramouli and Jun Yang. *End-to-End Support for Joins in Large-Scale Publish/Subscribe Systems.* In Proceedings of the 34^{th} International Conference on Very Large Data Bases (VLDB '08), Auckland, New Zealand, August 2008.
- Badrish Chandramouli, Jun Yang, Pankaj K. Agarwal, Albert Yu, and Ying Zheng. *ProSem: Scalable Wide-Area Publish/Subscribe.* In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08), Vancouver, Canada, June 2008 (demonstration).
- Badrish Chandramouli, Jeff M. Phillips, and Jun Yang. *Value-Based Notification Conditions in Large-Scale Publish/Subscribe Systems.* In Proceedings of the 33^{rd} International Conference on Very Large Data Bases (VLDB '07), Vienna, Austria, September 2007.
- Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. *Query Suspend and Resume.* In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07), Beijing, China, June 2007.
- Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. *On Suspending and Resuming Dataflows.* In Proceedings of the 23^{rd} International Conference on Data Engineering (ICDE '07), Istanbul, Turkey, April 2007 (poster paper).
- Badrish Chandramouli, Junyi Xie, and Jun Yang. *On the Database/Network Interface in Large-Scale Publish/Subscribe Systems.* In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06), Chicago, Illinois, June 2006.
- Badrish Chandramouli, Jun Yang, and Amin Vahdat. *Distributed Network Querying with Bounded Approximate Caching.* In Proceedings of the 11^{th} International Conference on Database Systems for Advanced Applications (DASFAA '06), Singapore, April 2006.

- Liangzhao Zeng, Hui Lei, and Badrish Chandramouli. *Semantic Tuplespace.* In Proceedings of the 3^{rd} International Conference on Service-Oriented Computing (ICSOC '05), Amsterdam, Netherlands, December 2005.

- Badrish Chandramouli, Hui Lei, Kumar Bhaskaran, Henry Chang, Michael Dikun, and Terry Heath. *Pushing the Envelope of Pervasive Access.* In Proceedings of the 2005 IEEE International Conference on Pervasive Services (ICPS '05), Santorini, Greece, July 2005.