Event-Driven Database Information Sharing

Luis Vargas, Jean Bacon, and Ken Moody

University of Cambridge, Computer Laboratory
{firstname.lastname}@cl.cam.ac.uk

Abstract. Database systems have been designed to manage business critical information and provide this information on request to connected clients, a passive model. Increasingly, applications need to share information actively with clients and/or external systems, so that they can react to relevant information as soon as it becomes available. Event-driven architecture (EDA) is a software architectural pattern that models these requirements based on the production of, consumption of, and reaction to events. Publish/subscribe provides a loosely-coupled communication paradigm between the components of a system, through many-to-many, push-based event delivery. In this paper, we describe our work integrating distributed content-based publish/subscribe functionality into a database system. We have extended existing database technology with new capabilities to realise EDA in a reliable, scalable, and secure manner. We discuss the design, architecture, and implementation of PostgreSQL-PS, a prototype built on the PostgreSQL open-source database system.

1 Introduction

Organisations invest in information technology to realise benefits through lower business costs, better information and better communication. Through the years, they have moved most of their critical data into databases, and automated many business processes using a variety of applications. Database systems and applications have been deployed incrementally to satisfy the needs of some particular area or *domain* of the business (e.g. a company branch). Domains are autonomous; each administers its own resources independently of others. With the passage of time, the number of databases, applications, and domains has multiplied. What once were closely controlled environments have evolved into large-scale information spaces that are both highly distributed and dynamic. Within such a domain-structured environment, the active sharing of information has become vital for the organisation's success. This applies not only within a domain, but also between different domains of the same organisation (and increasingly others).

To meet the need for active information sharing, domains often implement a large set of targeted tools. Different types of information are captured at different places (e.g. databases and applications), with different tools, each using its own propagation mechanism, and implementing its own method of consumption at the destination. Developers and database administrators must become proficient in all these tools, and the system must be able to support them all at runtime. Adding more applications to such an environment becomes a major ordeal.

A. Gray, K. Jeffery, and J. Shao (Eds.): BNCOD 2008, LNCS 5071, pp. 113–125, 2008.

Databases are an obvious point to implement active information sharing, since they maintain most of a business's critical information and reflect its current state. For example, an application may cause a change in a database, which can signal an event that is of interest to other applications within the same domain, or in other domains. Similarly, the receipt of an event by an application often results in a change in database state within its domain to record the event persistently. Such a style of interaction is best modelled by an event-driven architecture (EDA) [1]. In EDA, all interactions between components in a distributed system build on the production of, consumption of, and reaction to events. In parallel with the emergence of EDA, publish/subscribe middleware [2] has been designed and deployed, providing a popular communication paradigm for event-driven distributed systems. Publish/subscribe realises many-to-many, push-based delivery of events between loosely-coupled components. In this paper, we describe our work integrating distributed content-based publish/subscribe functionality into a database system to realise EDA in a reliable, scalable, and secure manner.

We complete this Section with a motivating scenario. Section 2 sets up the background in EDA, the publish/subscribe paradigm, and the PostgreSQL database system [3]. Section 3 discusses the design of PostgreSQL-PS, a database system enhanced with publish/subscribe functionality. The architecture of the system is described in Section 4. Section 5 discusses how events are distributed between multiple connected databases. The system's programming interface is presented in Section 6 and its implementation is discussed in Section 7. Section 8 outlines related work, and Section 9 concludes the paper.

1.1 Motivating Scenario

Consider a large-scale financial services firm. The firm has offices in different cities across America and Europe, see Figure 1. Each office (domain) is autonomous, and maintains a database system that stores its critical data and a set of applications, e.g. for automated trade processing. In such a scenario, most interaction lies within a domain, but there is also a need for inter-domain communication. For example, while some data feeds used to publish the latest stock

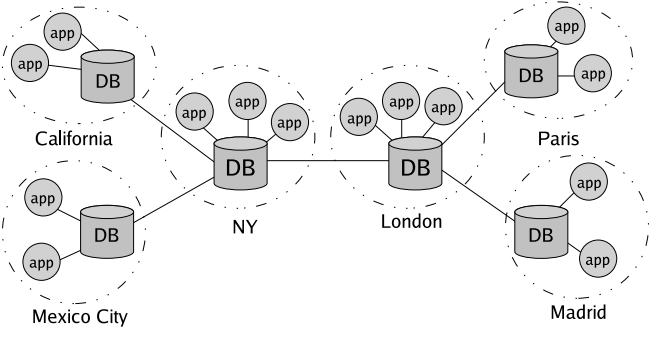

Fig. 1. A large-scale financial services firm

prices are made available only locally, others are also sent to external offices. Trading applications are specialised and thus expect different subsets of the available data feeds, e.g. based on the stock symbol, price, risk level, or combinations. Besides general stock processing, many other processes continuously keep track of specific situations, e.g. when a stock's minimum price-earnings ratio (PER) falls below a threshold. There will be a number of systems, both within and outside the local office, that require to be notified of such events, e.g. for real-time risk analysis.

Notice that information sharing in the scenario is inherently event based. To remain competitive, the firm requires an event-driven infrastructure to integrate the applications distributed across the various offices, and to support the active sharing of information. We believe that such infrastructure can be built efficiently on the database system. Firstly, database changes frequently trigger events. Secondly, for scalability, security, or simplicity purposes, business logic associated with event processing is often moved to the database (as stored procedures or triggers). Lastly, events must be logged in the database for reporting and audit.

2 Background

In this Section we establish the background on event-driven architecture (EDA), the publish/subscribe paradigm, and the PostgreSQL database system.

2.1 Event-Driven Architecture (EDA) and Publish/Subscribe

EDA [1] is an architectural pattern built on the production, detection, consumption of, and reaction to events. An event is defined as a happening of interest in the system. Events are generated by *event producers*. Based on its characteristics, an event is delivered to one or more *event consumers*. Consumers process the event and optionally execute an action. The architecture is loosely-coupled as producers and consumers do not require any knowledge about each other.

Publish/subscribe [2] is an asynchronous many-to-many event-based communication paradigm. In publish/subscribe, an *event client* may be an event producer (publisher), an event consumer (subscriber), or both. Event producers advertise the event types they will publish, and publish events, needing no knowledge of the subscribers. An event consumer specifies a set of subscriptions on events of interest. The event communication substrate, comprising one or more *event brokers*, accepts events from publishers and notifies them to subscribers whose subscriptions match. In a large-scale environment an event broker may serve a subset of the clients in the environment, for example, being associated with an administrative domain. Event brokers cooperatively distribute (route) events, while attempting to exploit locality and contain system complexity. A number of strategies for distributed event dissemination are discussed in [4].

Publish/subscribe comes in two flavours: *topic-based* and *content-based*. In topic-based publish/subscribe, events are published under a topic and consumers subscribe to that topic. In content-based publish/subscribe, event types are

defined as comprising typed attributes. A subscription includes a filter expression indicating attribute values of interest. Events with content that matches the filter expression are delivered to the appropriate consumers.

2.2 PostgreSQL

PostgreSQL [3] is an open-source object-relational database system written in C. Domain, referential, and transactional integrity, as well as multi-version concurrency control, are offered as some of its features. Active functionality is provided in the form of triggers and active rules. Because its operation is catalogue-driven, PostgreSQL can be extended, for example by adding new data types and functions. Functions can be written in C, Python, or procedural SQL (PgSQL).

3 PostgreSQL-PS Design

In this section we discuss the design of PostgreSQL-PS, a database system enhanced by distributed content-based publish/subscribe functionality. We describe the EDA aspects to be supported and establish requirements for the system.

3.1 EDA Aspects

The design of the PostgreSQL-PS system covers the four aspects of EDA: event publication, subscription, consumption and distribution. The first three relate to the role of the database system as a publish/subscribe event client. The fourth delegates to the database system the role of event broker.

Event Production: to define events and specify conditions for their generation.

Event Subscription: to define an interest in receiving specific event instances.

Event Consumption: to define local actions to be executed when events matching a subscription are notified.

Event Distribution: to receive events from external parties, e.g. applications and other database systems. Events (either received or generated at the database system) should be delivered to the relevant subscribers. Multiple interconnected database systems should cooperate to route events between locally-connected applications and applications connected to remote database systems.

3.2 Requirements

In the design of PostgreSQL-PS we have considered the following requirements: expressiveness, reliability, scalability, access control and operational simplicity.

Expressiveness. The event model must support fine-grained subscriptions.

Reliability. The system must provide guarantees regarding its operation. We focus on two aspects: transactional semantics and guaranteed event delivery.

Transactional Semantics
Event production. We must ensure that events are produced only by committed operations to avoid *dirty reads* by event consumers. We therefore need to defer the publication of an event until its triggering transaction has committed. Further, we must guarantee the production of events from committed transactions.

Event consumption. We must guarantee the *ACID* consumption of events. In particular, the execution of event-processing actions must be transactional.

Guaranteed Event Delivery. In general, the delivery of an event must be guaranteed despite failures between producer and consumer. Specifically, a consumer must (eventually) receive each event exactly once. Events from the same producer must be delivered to a consumer in the same order in which they were published.

Scalability. System performance must degrade gracefully with the number of clients, subscriptions, and events. For a single database system this requires an efficient mechanism for matching events against subscriptions. For multiple connected database systems, it also entails efficient event distribution.

Access Control. Each database system must be able to control which clients can publish and subscribe to each event type.

Operational Simplicity. The database system must provide an integrated view of database and publish/subscribe operations through a simple interface.

4 PostgreSQL-PS Architecture

In this Section we describe the different components of the PostgreSQL-PS system architecture. All these components are defined in the context of a database. Multiple databases, each having different instances of these components, can be hosted within the same distributed database environment.

4.1 Event Types

Event types are used to structure the event space, each event being an instance of an event type. An event type has a system-wide unique name [5] and a schema that describes it. The schema is a set of attribute-name, data-type pairs. Valid data types are the native types defined by the SQL92 specification [6] (e.g. varchar, int, datetime). Event types are stored in the database system catalogue. They are used to verify that a) an event instance conforms to its type schema, b) a subscription filter refers to existing attributes, and c) functions and operators in the filter are valid for the attribute types.

4.2 Events

An event is a set of attribute name/value pairs conforming to an event type schema. In the database it is represented as a tuple structure. Events generated at the database have two properties: *visibility* and *reliability*. The visibility of

an event determines when the event is published with regard to the transaction triggering the event. It can be either *immediate* or *deferred*. In the former, the event is published as soon as it is generated. In the latter, the publication of the event is deferred until its triggering transaction has committed. The reliability of an event determines whether its delivery is *non-guaranteed* or *guaranteed*. In the former, events are delivered at-most-once. In the latter, events are delivered exactly-once, ordered with respect to each producer.

4.3 Subscriptions

A subscription expresses interest in consuming (a subset of) events of some type. Subscriptions are named and specify an event type, an optional filter, and a source. The filter is a SQL predicate over the event type's attributes and, possibly, stored data. A large number of built-in operators and functions can be specified as part of the filter. The source of a subscription can be *external* or *internal*. An external subscription is issued by a client application or received from another database. An internal subscription is defined at the database to process local events. Subscriptions have a *local* or *global* scope. A local subscription only applies to events known to the local database. A global subscription also expresses interest in events known to other databases (directly or indirectly) connected to the database. Subscriptions are persistently stored in the database system catalogue, in order to survive system failures and client disconnections.

4.4 Queues

Queues contain events. For each event type there are three queues: *in*, *out*, and *exception*. Events locally produced or received from external parties are enqueued in their corresponding *in-queue*. Events that have been matched against subscriptions are enqueued in their *out-queue* for their delivery (external subscriptions) or local processing (internal subscriptions). Events for which processing fails are enqueued in their *exception queue*. Each in- and out- queue has two instances, *non-persistent* and *persistent*. Non-persistent queues are volatile data structures that hold non-guaranteed events in memory. Persistent queues use database storage to store guaranteed events reliably on disk. They are implemented as special tables with no `INSERT`, `UPDATE`, `DELETE`, or trigger statements. Events stored in a queue can be consulted via `SELECT` queries on the event schema and additional system information (e.g. enqueue time). Persistent queues can be *non-auditable* or *auditable*. In the first (default) case, an event is deleted from a queue when it is no longer required (e.g. it has been successfully delivered to a consumer). In the second, the event is retained in the queue for auditing purposes.

4.5 Advertisements

Advertisements are either directly created at a database or introduced by a local application. A database must advertise an event type before it can produce or distribute events of that type. Advertisements are stored persistently in the database system catalogue.

4.6 Links

A link represents a connection to a remote database. It specifies contact information and associated authentication data. On startup, a database connects to all its defined links. Advertisements, global subscriptions, and events are propagated through these links. A connected set of databases forms a distributed system that actively shares information as described in the next Section.

5 PostgreSQL-PS Cooperative Event Distribution

In this Section we describe the mechanism used to distribute events cooperatively between connected databases. In the current prototype, we consider a peer-to-peer interconnection model in which databases communicate symmetrically in an acyclic topology. In practice databases can be connected in any way, provided that we identify a spanning tree for routing purposes. Factors to consider when connecting two databases include: administrative constraints, knowledge about the locality of consumers (or producers) of events of interest, and network latency.

Events are cooperatively distributed using an *advertisement-based* filtering scheme [7]. In this scheme, databases build event dissemination trees by propagating advertisements and subscriptions as follows:

1. An advertisement for an event type is propagated by following every database link. Each database stores advertisements received from the previous database. This builds, for the event type, a dissemination tree from the advertising database to every other database.
2. A subscription is propagated by reversing the links of databases with stored advertisements for that type. Each database stores the subscription received from the previous database. This builds, for the event type, a dissemination tree from the subscribing database to every database that produces events of that type.

Events are distributed by following the links of databases with stored subscriptions that match their type and content. After receiving an event, a database evaluates 1) the set of internal subscriptions and external subscriptions issued by local client applications, and 2) the external subscriptions received by linked databases. If no subscription matches the event, it is discarded.

We illustrate event distribution in Figure 2. For clarity we consider a single event type `t`. We show six connected databases $\texttt{DB}_{1-6}$ and a set of applications $\texttt{App}_{1-6}$. $\texttt{App}_1$ and $\texttt{DB}_2$ produce events of `t`, and $\texttt{App}_3$ and $\texttt{DB}_6$ are event consumers.

First, $\texttt{App}_1$ advertises `t` with $\texttt{a}_1$, via its local database $\texttt{DB}_1$. $\texttt{a}_1$ is propagated to, and stored by $\texttt{DB}_{2-6}$. Then $\texttt{DB}_2$ creates the advertisement $\texttt{a}_2$ which is propagated to, and stored by $\texttt{DB}_3$, the only `DB` with a new source of events for `t`.

Next, $\texttt{DB}_6$ creates the global subscription $\texttt{s}_6$, which by reversing the paths of $\texttt{a}_1$ and $\texttt{a}_2$, is propagated to, and stored by $\texttt{DB}_4$, $\texttt{DB}_3$, $\texttt{DB}_1$, and $\texttt{DB}_2$. On request of $\texttt{App}_3$, $\texttt{DB}_3$ creates the global subscription $\texttt{s}_3$, which is propagated to, and stored by $\texttt{DB}_1$ and $\texttt{DB}_2$, extending the dissemination route for `t`. When $\texttt{DB}_1$ receives the event $\texttt{e}_1$ from $\texttt{App}_1$, it is propagated to $\texttt{App}_3$ and $\texttt{DB}_6$.

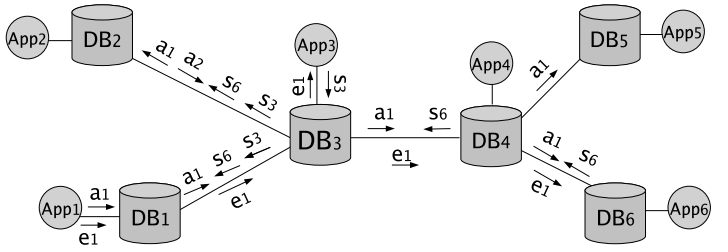

Fig. 2. Cooperative Event Distribution

6 PostgreSQL-PS Programming Interface

PostgreSQL-PS provides two interfaces to programmers. A *database programming interface* supports system administration and database-side event processing. An *application programming interface* is available to client applications.

6.1 Database Programming Interface

This interface, see Table 1, extends SQL with a number of publish/subscribe related statements. It is accessible from the database system console (Psql), as well as from client-level interface implementations such as JDBC.

An event type is created using the `CREATE EVENT TYPE` statement. This statement also creates in, out, and exception queues for the event type. `ALTER QUEUE` sets the auditable behaviour of a queue. An event type must be advertised before events of that type can be published or subscribed to. This is done using `ADVERTISE`. Events are generated at the database using `PUBLISH`; the statement is parametrised with the event *visibility* and *reliability*. `PUBLISH` can be used as a separate statement or within a transaction. It can also be set as the *action* of an active rule. This automates the production of events after data manipulation commands, possibly referring to the transition tables `NEW` and `OLD`. A database

Table 1. Database Programming Interface

CREATE EVENT TYPE *event_type* AS (*att1 datatype, att2 datatype, ..*)
ALTER QUEUE *queue_name* SET [NON-AUDITABLE|AUDITABLE]
ADVERTISE *event_type*
PUBLISH [IMMEDIATE|DEFERRED] [NON-GUARANTEED|GUARANTEED] *event_type* (*attvalue1, attvalue2, ..*)
CREATE RULE *rule_name* AS ON {INSERT|UPDATE|DELETE} TO *table* [WHERE *filter*] PUBLISH *event_type*(*attvalue1, attvalue2, ..*)
CREATE [LOCAL|GLOBAL] SUBSCRIPTION *sub_name* ON *event_type* [WHERE *filter*] EXECUTE *func_name*(*args*) [WITH *priority*]
CREATE LINK *link_name* TO *address port* USING *user password*
GRANT [PUBLISH|SUBSCRIBE] ON EVENT TYPE *event_type* TO {*user*|*role*}

subscribes to events of some type using `CREATE SUBSCRIPTION`; the statement is parametrised with the subscription scope. As an optional filter, an SQL predicate can be specified via a `WHERE` clause on attributes of the event type, as well as on stored data. Subscriptions created at the database are always *internal*, and must specify a function to process received events. Functions can be written in any of the languages supported by the database, allowing developers to focus on the most suitable for a given task (e.g. PgSQL for data-centric operations or C for computationally intensive logic). The way in which a function is passed an event depends on its implementation language, e.g. as a pointer to an `Event` structure in C, or a top-level `RECORD` variable in PgSQL. An optional priority can be assigned to the subscription, an absolute value that determines the order of evaluation for subscriptions to a given event type. A link is defined using `CREATE LINK`, which specifies the address and port on which a remote database services publish/subscribe connections, and an authorised user and password in that database. There are `DROP` statements for `EVENT TYPE`, `RULE`, `SUBSCRIPTION`, and `LINK`, as well as an `UNADVERTISE` statement. Privileges on each of these statements can be assigned and removed from database users and roles using `GRANT` and `REVOKE`. Information about existing publish/subscribe-related objects (e.g. event types, subscriptions, and links) is made available through restricted catalogue views.

6.2 Application Programming Interface (API)

This interface, depicted in Table 2, allows applications to access the database system publish/subscribe functionality. We provide a Java implementation of the API. In this, functions are supported via a `Client` object as described next.

An application connects to the database system using the `Client connect` method. This requires the address and port where the database services publish/subscribe clients. A valid user and password are needed to authorise the client, and to associate any stored subscriptions to the connection. Event types are created by instantiating the `EventType` class. This class contains a name, and a *Map* of two attributes: name and type. Valid types are *String*, *Date*, and all subclasses of *Number*. The API translates these types to SQL92 data types when an application requests the database to advertise an event type using the `advertise` method. Events are created by instantiating the `Event` class. This class has an associated event type and a *Map* of two attributes: name and value. The value is an object of the corresponding attribute type. Events are published, with the requested reliability, via `publish`. A subscription is issued, with the specified scope, via `subscribe`. An optional filter can be defined via

Table 2. Application Programming Interface

`Client.connect(`*address, port, user, password*`)`
`Client.advertise(`*EventType*`)`
`Client.publish(`*reliability, Event*`)`
`Client.subscribe(`*sub_name, EventType, scope, filter, Callback*`)`

an SQL predicate. Finally, a class implementing the `Callback` interface must be specified to process any events received. The API keeps a persistent *Map* of subscriptions and callbacks. Events from the database are piggybacked with the matched subscription name. Based on the SQL92 data types of attributes in events received, the programming interface builds an `Event` Java object that is passed to the callback class.

7 PostgreSQL-PS Implementation

We have implemented PostgreSQL-PS by extending the PostgreSQL [3] 8.0.3 code base. We chose PostgreSQL for its rich set of features and the ability to analyze and extend its source code. We now describe the PostgreSQL-PS process architecture and discuss how we fulfil the requirements established in Section 3.2.

7.1 Process Architecture

Figure 3 shows the PostgreSQL-PS process architecture. The various PostgreSQL components that are reused are shown on the left.

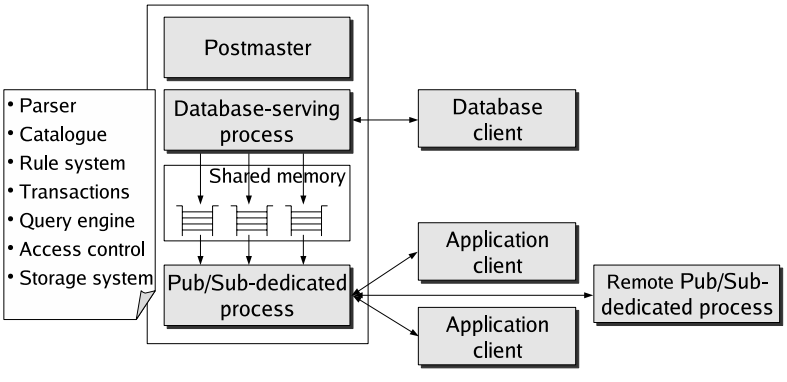

Fig. 3. PostgreSQL-PS Process Architecture

As in PostgreSQL, a Postmaster listens on a well-known port and forks a new database-serving process for each database client TCP connection. These clients, usually Psql or JDBC-based, are served using synchronous request-reply. The PostgreSQL parser was extended to provide clients with the database publish/subscribe programming interface. A process dedicated to publish/subscribe handles TCP connections from publish/subscribe clients and remote databases. The Postmaster forks this process on startup. Communication with publish/subscribe clients is message-oriented and asynchronous, via non-blocking sockets. We notify the publish/subscribe process of relevant operations (e.g. events and subscriptions) issued from a database-serving process; for this we use a set of control queues in a shared memory segment. Database-serving processes enqueue notifications that are dequeued and analysed by the publish/subscribe process.

7.2 Transactional Event Production and Consumption

We currently implement two event visibility and reliability combinations, *immediate non-guaranteed* and *deferred guaranteed*, as we expect them to cover most application scenarios. For the former, a published event is enqueued in its non-persistent in-queue. There is no dependency between the transaction producing the event and the publish operation. When the event is dequeued from its in-queue, it is matched against subscriptions and immediately sent to its consumers, via its non-persistent out-queue. For the latter, a publish operation enqueues the event in its persistent in-queue. This is done within the running transaction to ensure the atomicity of publish with other operations. If enqueuing fails, the transaction rolls-back and the user is notified of the error. Otherwise the event, together with the ID of its producing transaction, is stored in the queue. When the transaction commits, a notification containing the transaction ID is enqueued in a control queue. On-commit hooks [8] have been incorporated for this purpose. When the publish/subscribe process dequeues the notification, it matches events with the notified transaction ID against subscriptions. For each matched consumer, one instance of the event is enqueued in the event's persistent out-queue for processing or delivery. Once matching is completed, the event is removed from its persistent in-queue. The insertion of the event instances in the out-queue and the removal of events from the in-queue are executed within the same transaction.

On consumption, the dequeue of an event from its persistent out-queue and the execution of its processing function take place within a transaction. The execution of multiple functions for the same event is serial. Functions are executed in turn, in separate transactions, according to subscription priority. This ensures atomicity of function execution and isolation of execution for independent functions. If processing an event fails (e.g. due to a violated constraint), the event is removed from its out-queue and enqueued in its exception-queue with a description of the error. These two operations are executed in a single transaction.

7.3 Guaranteed Event Delivery

We ensure exactly-once ordered delivery of events between a sender and a receiver on a direct connection using an acknowledgement-based protocol with unbounded sequence numbers [9]. At the application side, the protocol is handled automatically by the API. Sender S keeps a sequence number s that is incremented for each event e to be sent to receiver R. Before sending $e[s]$, i.e. event e with sequence number s, S persistently stores it with an associated timestamp. We denote as $S.e[s]$ the event e with sequence number s sent by S. To enforce ordering, a receiver keeps an array N of sequence numbers, one for each sender. $N[S]$ denotes the sequence number associated with sender S. N is stored persistently so it survives receiver failures. On receipt of an event $S.e[n]$: if $n < N[S]$, R acknowledges n to S and discards the event. If $n = N[S]$, R processes the event, increments $N[S]$, and acknowledges n to S. If $n > N[S]$, R acknowledges $N[S]$ to S and discards the event. The sender can delete $e[s]$ when

it receives an acknowledgement for s. If S does not receive an acknowledgement for $e[s]$ within a predefined timeout, it resends using an *exponential backoff*. In this, the timeout starts at 4 seconds and doubles at each retry up to a maximum of 64 seconds.

7.4 Scalability

Publish/Subscribe-related data indexing: At a database we need to quickly retrieve 1) the event type schema used to validate an event, 2) the in, out, or exception queue where an event must be stored, and 3) the set of subscriptions to be evaluated against the event. Fetching this data from disk every time would damage database performance. Therefore, we cache and index publish/subscribe-related data in main memory. A hash table indexes event types by name: each bucket stores the schema of the event type, its associated queues, and a pointer to a dynamic array of subscriptions. Because of sequential locality, this structure allows efficient iteration over the set of subscriptions for a given event type.

Execution Plan Caching for Subscription Filters: When the database receives a subscription, its filter is parsed and translated into an *execution plan*. When the subscription is evaluated, the database query engine needs only to execute this plan. Parsing and planning is thus performed only once, instead of at each evaluation. The performance gain is more significant if the subscription filter refers to stored data, as planning of queries on tables takes more time.

Logical Event Deletes: An event in an in-queue is deleted after it has been processed locally and matched against subscriptions. An event in an out-queue is deleted after it has been acknowledged. In PostgreSQL a `DELETE` operation is logical, i.e. it does not physically remove a tuple from disk. To reclaim the space of deleted tuples, a separate `VACUUM` operation is used. This approach reduces the time to delete an event from a queue, at a cost in disk space utilisation. We expect that all queues in the database are vacuumed periodically (e.g. once a day at a low-usage time), with more frequent vacuuming of heavily updated queues.

8 Related Work

"Queues are databases" was stated more than ten years ago [10]. Accordingly, some vendors have incorporated message queuing into their database systems. SQL Server Service Broker [11] provides asynchronous and reliable dialogues between databases to support distributed applications. Communication is bi-directional between two databases; publish/subscribe is not supported. Oracle Streams [12] supports one-to-many asynchronous replication via multi-consumer queues. Replicas can specify content-based rules to propagate only a subset of data changes from the master database. Rules are not global (they must specify a source and a destination queue) so that a replica cannot express interest in data beyond its master, unlike PostgreSQL-PS's global subscriptions.

9 Conclusions

Maintaining most of a business's critical information and reflecting the state of its daily processes, database systems are in a cardinal position to support active information sharing. EDA provides an appropriate model for active data sharing based on the production and consumption of events. Publish/subscribe is a suitable loosely-coupled communication paradigm. Integrating distributed content-based publish/subscribe functionality into the database system is therefore a promising approach to active information sharing. On one hand, databases already provide many features that an event-driven architecture can exploit, such as persistent storage, transactions, and active rules. On the other hand, integrating publish/subscribe into the database system leads to information-sharing systems that are simpler to deploy and maintain. We are currently evaluating PostgreSQL-PS against a decoupled database - publish/subscribe system. Preliminary experimental results show that the execution of functions that require to access the database frequently, e.g. logging events or evaluating subscriptions that refer to tables, is faster in PostgreSQL-PS. However, the evaluation of subscriptions that refer only to event content is, on average, slower, as the query engine incurs an additional overhead. We are thus planning to incorporate an event/subscriptions matching algorithm that pre-filters subscriptions based on event content and employs the query engine only as needed.

References

1. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, Reading (2002)
2. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The Many Faces of Publish/Subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
3. The PostgreSQL Global Development Group (2008), `www.postgresql.org`
4. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer, Heidelberg (2006)
5. Pesonen, L.I.W., Bacon, J.: Secure Event Types in Content-Based, Multi-domain Publish/Subscribe Systems. In: Proc.of the 5th International Workshop on Software Engineering and Middleware, pp. 98–105 (2005)
6. American National Standards Institute: Standard x3.135-1992 (1992)
7. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Tran. on Computer Systems 19(3), 332–383 (2001)
8. Paton, N.W., Díaz, O.: Active Database Systems. ACM Computing Surveys 31(1), 63–103 (1999)
9. Comer, D.E.: Internetworking with TCP/IP vol II. ANSI C Version: Design, Implementation, and Internals. Prentice-Hall, Englewood Cliffs (1998)
10. Gray, J.: Queues are Databases. In: Proc. of the 7th High Performance Transaction Processing Workshop (1995)
11. Aschenbrenner, K.: SQL Server 2005 Service Broker. Apress (2007)
12. Oracle: 11g Streams Replication Administrator's Guide (2007)