Wolfgang Lehner

Data Management Support for Notification Services

Erstveröffentlichung in / First published in:

Theo Härder, Wolfgang Lehner, Hgg., 2005. *Data Management in a Connected World*. Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.

DOI: https://doi.org/10.1007/11499923_7

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

Data Management Support for Notification Services

Wolfgang Lehner

Technische Universität Dresden, Germany
lehner@inf.tu-dresden.de

Abstract. Database management systems are highly specialized to efficiently organize and process huge amounts of data in a transactional manner. During the last years, however, database management systems have been evolving as a central hub for the integration of mostly heterogeneous and autonomous data sources to provide homogenized data access. The next step in pushing database technology forward to play the role of an information marketplace is to actively notify registered users about incoming messages or changes in the underlying data set. Therefore, notification services may be seen as a generic term for subscription systems or, more general, data stream systems which both enable processing of standing queries over transient data. This article gives a comprehensive introduction into the context of notification services by outlining their differences to the classical query/response-based communication pattern, it illustrates potential application areas, and it discusses requirements addressing the underlying data management support. In more depth, this article describes the core concepts of the *PubScribe* project thereby choosing three different perspectives. From a first perspective, the subscription process and its mapping onto the primitive publish/subscribe communication pattern is explained. The second part focuses on a hybrid subscription data model by describing the basic constructs from a structural as well as an operational point of view. Finally, the *PubScribe* notification service project is characterized by a storage and processing model based on relational database technology.
To summarize, this contribution introduces the idea of notification services from an application point of view by inverting the database approach and dealing with persistent queries and transient data. Moreover, the article provides an insight into database technology, which must be exploited and adopted to provide a solid base for a scalable notification infrastructure, using the *PubScribe* project as an example.

1 Introduction

The technological development in recent years has created an infrastructure which enables us to gather and store almost everything that can be recorded. However, the stored

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

data gets frequently lost in the existing varieties of databases. The main reason is that nobody is willing to browse multiple databases and specifically search these databases for certain entries.

Multiple methods from a database-technological point of view are trying to leverage this very general problem. The context of information integration [26] tries to come up with an either logically (multi-database systems or virtual database systems) or physically integrated data set, often seen in data warehouse environments [16]. From a more application-oriented point of view, methods and techniques coming from the area of knowledge discovery in databases try to generate hypotheses which might be of some interest for the users of the data set. The real benefit of this approach is that a user may specifically focus on these results as a starting point for an interactive analysis [10, 19, 9].

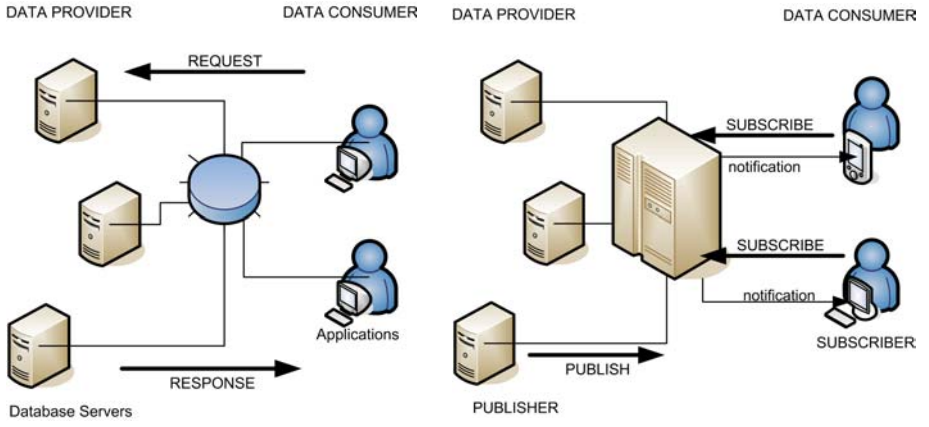
A completely different approach is taken when inverting the current way of interacting with databases by moving from a *system-centric* to a *data-centric* behavior [22, 23]. The query-based approach follows the request/response paradigm, where the user (or client) is posing a query and the (database) system tries to execute the query as fast as possible. The result is delivered via basic interfaces (like ODBC, JDBC, or CLI) to the user's application context. Fig. 1a illustrates this interaction pattern with database management systems on the one side acting as data providers and clients on the other side acting as data consumers. Both parties are in a close relationship with each another, i.e. every client has to know the location and the context of the specific database.

1.1 Publish/Subscribe as the Base for Notification Systems

Inverting the "request/response" idea leads to the communication pattern very well known as "publish/subscribe" [5], which is used in various situations. For example, publish/subscribe is utilized in software engineering as a pattern [12] to connect individual components. In the same vein, publish/subscribe may be used to build a data-centric notification service. Notification services consist of publishers, subscribers, and finally, a (logically single) notification brokering system. Fig. 1b gives a conceptual overview of the scenario. On the one side, publishing systems (or publishers) are acting as data providers, generating information and sending data notifications to the brokering component as soon as the information is available. Notifications are collected, transformed into a pre-defined or pre-registered schema and merged into a global database. Depending on the type of subscriptions (section 3.1), notifications may remain in the database only for the time span required to notify interested subscribers.

On the other side, subscribers—acting as data consumers—are registering "interest" (information template, profile, ...) providing the delivery of certain notifications using specific formats and protocols with a pre-defined frequency. The notion of interest is manifold and will be further discussed in the following section. Furthermore, query specification and result transmission are decoupled from a subscriber's point of view. Once a query is formulated, the user is no longer in contact with the notification system but receives a notification only if new messages of interest have arrived at the database system or if a given time interval has passed. The advantage for the user is tremendous:

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7



a) "request/response" paradigm b) "publish/subscribe" paradigm

Fig. 1 Comparison of "Request/Response" and "Publish/Subscribe" Paradigm

once a query (in form of a subscription) is specified, the user does not have to wait for the answer of the query, because the result will be delivered automatically according to the pre-defined delivery properties.

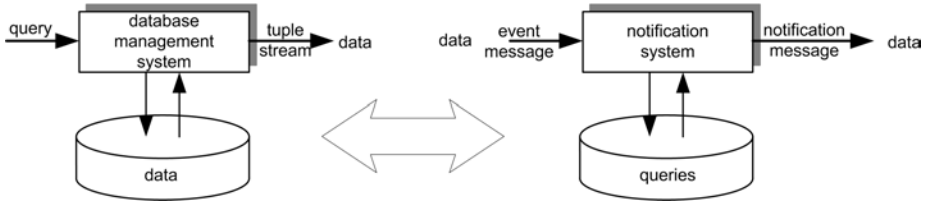
Obviously, publisher and subscriber are roles associated with certain applications or (human) users implying that, for example, received notification messages may be forwarded to another notification system. In this scenario, a single component is then acting as subscriber and publisher at the same time. The benefit of notification services from an application point of view consists in the following facts:

- Data providers and data consumers are decoupled and do not know each other—the connection is purely data-driven.
- Profiles articulating a subscriber's interest allow (depending on the current system) a very detailed specification of the requested piece of information.
- Information is delivered only if certain delivery criteria are fulfilled. Therefore, notification systems may be regarded as a core mechanism to tackle the problem of a general information flood.

1.2 Data-Centric Versus System-Centric Data Delivery

From a database point of view, notification services exhibit properties which have a dramatic impact on the way data is treated and queries are executed. Fig. 2 illustrates the basic principle of the different prerequisites compared to the request/response-driven querying model. While database queries are executed within transactions and therefore isolated from each other, the notification evaluation queries are now clustered together and executed simultaneously [27, 20], thus decreasing the overall query run time and

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7



a) "request/response" paradigm b) "publish/subscribe" paradigm

Fig. 2 Query-driven vs. Data-driven Execution

increasing the overall capacity of a notification system. Obviously, handling thousands of subscriptions within a single system requires specific support from a database system [30, 3, 20].

To put it in a nutshell, the main difference from a database point of view consists in the fact that data structures no longer reflect the main object of management. In contrast, queries are now efficiently stored, indexed, and transformed, providing the basis for optimizations with regard to the set of standing queries, which have to be applied to the stream of incoming event messages. Furthermore, a notification system may comprise multiple local (and only partially integrated) schemas according to the registration of the publisher. Because a publisher may come and go, the set of local schemas is highly volatile, implying an integration process either on-the-fly during the registration of a standing query or partially by the user itself.

1.3 Application Scenarios

In most application scenarios, notification systems can be seen as an add-on and not as a full substitute for a regular query-driven database management system. Notification systems are used in applications requiring a pro-active propagation of information to specific consumers. In the following, three very different application scenarios are discussed:

- *News service*
  A very popular and already prospering service implementing a very limited kind of notification technique may be seen in news services, which send e-mails or short messages containing abstracts of various news articles on a regular basis (one of the first was [35] followed by many others like [25]). The functionality of news notification services highly varies from system to system. For instance, notifications could depend only on time stamps or time periods since the last delivery, or they might be based on the evaluation of specified predicates within subscription templates. For example, a notification can be generated if the stock price of IBM reaches a certain value. Simple news services are often synonymous to the "push service" of documents.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

- *Business intelligence applications*
  Combining the idea of a notification service and the concept of data warehousing [16] leads to the pro-active support of decision support systems. A data warehouse system provides an integrated, long-term, and logically centralized database to retrieve information necessary for decision support, supply chain management, customer relationship management, etc. Due to the nature of data warehousing, these database applications exhibit specific characteristics with regard to data volume, update characteristics, and aggregation-oriented organization of data. Information stored in a data warehouse are usually exploited by using pre-defined reports printed on a regular basis, by interactive exploration using standard OLAP tools, or by exportation into special statistical software packages . In such a scenario, for example, all sales orders are flowing through a notification system. The system keeps track of the sold products of a certain category within a specific region and can automatically re-order the required products. This application domain highly correlates with the attempt of building active data warehouse systems. Notification systems are a necessary requisite.

- *Production monitoring*
  Notification systems in the small (also called streaming systems) are able to monitor the manufacturing quality of a machine within a long assembly line. The streaming system has to ensure that incoming event notifications are analyzed and evaluated against the set of standing queries within a certain period of time. If tolerances are too high or if a pattern of irregular behavior within the assembly line is discovered, notification messages are sent out to slow or shut down some machines.

It can be seen that notification services, on the one hand, cover a broad spectrum of applications and, on the other hand, exhibit a strong impact on the evaluation strategies for answering multiple standing queries. Moreover, the application areas shown above motivate the classification of notification systems into the following two classes:

- *Subscription systems*
  A subscription system is used to evaluate a huge number of potentially complex-structured standing queries on incoming data. In comparison to the characteristics of a data stream system, each publication reflects an isolated action, i.e. publications of a single publisher are (except for the schema) not related to each other—which especially holds for their time stamps. Typical publications, for example, may consist of large XML documents [32].

- *Data stream systems*
  A data stream system is used to analyze continuous streams of data typically coming from data sensors. Publications usually arrive periodically and comprise single individual values (like temperature values).

In the following, we focus on subscription systems which typically require more advanced functionality from a modeling as well as an architectural point of view as proposed for example in [31]. For further information on application scenarios in the context of data stream technology, we refer our readers to [15, 2, 6, 4]. Notification systems and subscription systems are therefore used synonymously.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

1.4 Structure of the Contribution

The remainder of this contribution is focusing on the notification management system *PubScribe*, which aims to build a notification system solely based on the publish/subscribe communication pattern and exploits advanced database technology. Therefore, the following section outlines the basic characteristics of *PubScribe* in the context of a general architecture and communication pattern scheme. Section 3 then discusses the subscription data model used by *PubScribe*, which will be mapped to a processing model in section 4. Section 5 finally summarizes and closes the contribution with an outlook on further work to be done in this context.

2 General Architecture and Characteristics

In this section, we identify and explain a number of different characteristics, which apply to the family of notification systems, and provide a general overview of the components required to build a notification system.

2.1 General Architecture

A notification system basically consists of three major components (back-end, front-end, and brokering component), which can be seen in Fig. 3. In general, a notification system accepts event messages and cleans, transforms, and combines these messages with other data available to the notification system. Finally, individual notification messages are generated and delivered to the consumer, i.e. subscriber. The general structures and their functions are as follows:

- *Event message provider*
  The component of an event message provider accepts incoming event messages and transforms them into a shape which can be processed by the underlying notification engine (similar to wrapper technology as described in [26]). Usually, each message is decomposed into multiple rows spread over multiple relational database tables. In Fig. 3, an event message provider exists for XML documents, for the result of SQL statements (e.g. in the context of the execution of stored procedures), and for a service that periodically crawls specific web sites to retrieve data and propagate them to the notification system. The output of an event message goes into a message table, which is exploited by the subscription system.
- *Notification message delivery component*
  The notification message delivery component extracts the results provided by the notification engine and creates appropriate notification messages, personalized either for a human user or—according to a given message schema—for an application as consumer. Fig. 3 depicts three different delivery handlers to forward notification messages via the HTTP-protocol for web access, the SMTP-protocol for use of electronic mail, and a generic file protocol to store messages in a regular file.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

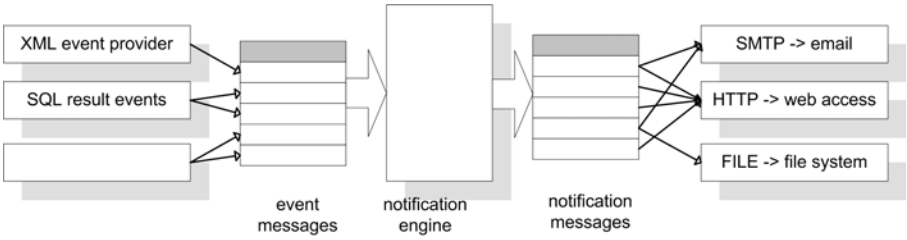


Fig. 3 General Architecture of a Notification System

More sophisticated push-based delivery methods require either an adequate network infrastructure or a cooperating client software. Whenever pushing data to a client is not directly supported by the underlying transport protocols (i.e. TCP, HTTP, ...), push services are implemented in an extended pull style. In this case, a specific piece of software is running in the background on the client side, permanently polling for new information, and thus, pretending a server push to the client. Such strategies are called smart pull or pull++. Another technique for simulating push is server-initiated pull. In this case, the server sends a short notification to the client stating that there is new data ready for delivery. The client then downloads the notification message using a regular pull operation. It is worth mentioning here that a notification service which is logically based on the publish/subscribe paradigm can be implemented using push as well as pull techniques for data delivery.

- *Notification brokering component / notification engine*
  The central piece of a notification system consists of the brokering component which is tightly integrated into an underlying database engine to efficiently answer the registered standing queries for incoming messages. Because the event message provider and the notification message delivery component are of little interest from a database perspective, we focus on the brokering component in the remainder of this contribution.

2.2 General Communication Pattern

Within the *PubScribe* system we pursue a communication pattern on two different levels. From a user (publisher and subscriber) point of view, the notification service consists of five service primitives, as depicted in Fig. 4.

In a very first step, publishers are requested to register their publications at the notification system (REGISTER primitive), which will set off the initiation of the appropriate event message provider and the creation of the appropriate message tables for this specific publishing component. Even more important, each publisher must submit a schema definition of the proposed messages. After registration, publishers use the publish/service primitive to submit event messages for further processing.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

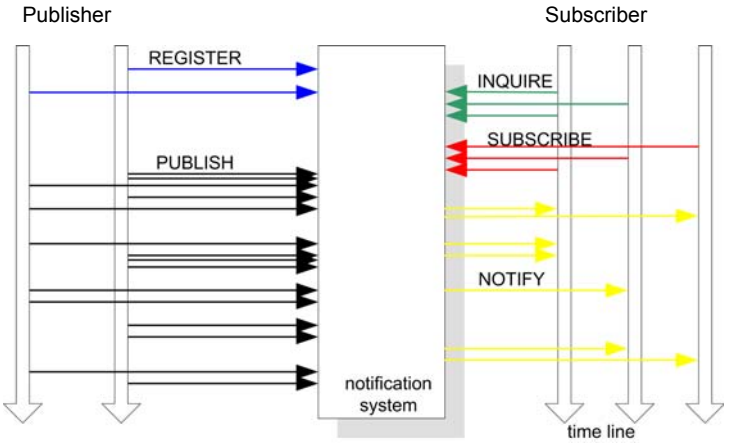


Fig. 4 Communication Pattern in Publish/Subscribe-Notification Systems

On the subscriber side, a potential user may issue an INQUIRE primitive to learn about publishers currently present and their local schemas. The consumer may then decide to place a subscription (standing query) based on their schemas including start, stop, and delivery conditions (section 3.1) using the SUBSCRIBE primitive. Once a notification message is ready, the notification message delivery component is using the NOTIFY primitive on the consumer side to deliver the result.

On a lower (communication) level, describing the interaction mechanisms of different components within a network of notification systems, the *PubScribe* system applies the publish/subscribe paradigm and maps the five service primitives on the application level to the publish/subscribe communication pattern. The REGISTER and INQUIRE primitives are mapped onto a so-called one-shot publish/subscribe pattern, implying that a subscription is only valid as long as a single notification message has not arrived. More interestingly, the PUBLISH and SUBSCRIBE/NOTIFY primitives are also translated into a publish/subscribe pattern. After registering, the *PubScribe* notification system subscribes at the publisher and places a subscription to ask for event messages. The PUBLISH primitive on the application level then corresponds to the PUBLISH primitive on the communication level. Similarly, the SUBSCRIBE primitive of a consumer is mapped to the SUBSCRIBE primitive at the lower level, and the NOTIFY primitive is treated as a PUBLISH primitive issued by the notification delivery component (taking on the role of a publisher with regard to the subscriber).

2.3 Classification of Notification Systems

As a final consideration with regard to the general perspective of notification systems, we provide some properties which might be used to classify the very broad set of notification systems. It is worth mentioning that these properties are not completely orthog-

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

onal to each other, i.e. certain combinations may not make that much sense from an application point of view.

Document-Based Versus Data-Stream-Based Notification Systems

The most distinctive feature of a notification system lies in the differentiation between document-based and data-stream-based characteristics. In the context of document-based systems, each message consists of an individual entity and is treated separately from every other message. Especially the "birth" of the message in a document-based environment is not related to the "birth" of other messages of the same publisher. This means that from an application point of view, there is no system-relevant correlation of the publication of individual messages. Typical examples of document-based notification systems are news tickers that report on current developments.

The other extreme is characterized by data streams. In this case, the publication of a message happens periodically in the sense that a message does not reflect an individual entity, but is comprised of on-going data either to complete or to bring the current state up-to-date. In the former case, data is added to the message, while in the latter case, data is overwritten, thus implying that the notification system holds the most current state with regard to some real-life object. An example for streaming systems is the control procedure of an assembly line, where sensors are reporting the current state of a machine on a periodic basis [7, 29].

PubScribe, which serves as an example within this contribution, is a classic representative of a document-based notification system. Streaming systems dealing with an infinite set of tuples are not discussed. The reader is referred to excellent literature like [15, 24, 3, 14] that focuses on this topic from an architectural point of view.

Time-Driven Versus Data-Driven Notifications

The second characteristic with regard to information delivery is the classification of the kind of "event" which has to happen in order for a notification to be sent out to the client. Notifications are either dispatched periodically after a specified amount of time or they are initiated due to a data-driven event. A typical example for the first case is to send out an electronic newsletter every day at 6 p.m. A new letter (or a collection of accumulated single news articles) is simply sent out after another 24 hours have passed. Alternatively, a subscriber may be interested in getting notified by the notification system aperiodically, i.e. only when a certain event occurs, e.g. a certain stock value passes a certain threshold. Data-driven events are usually connected to insert or update operations in the underlying database and result in aperiodic notifications. They are closely related to the trigger concept of active and relational database systems [21]. In practice, the combination of both notification modes is most interesting. For example, a user might want to be informed immediately if the IBM stock falls below a certain value, and additionally, get a weekly summary for its performance throughout the week.

Full Update Versus Incremental Update

For the subscription management system, it is important to know what to do with data which was already received by the client through a previous delivery. In case of a thin client like a simple web browser without any application logic and local storage capac-

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

ity, the server always has to deliver a full update to the client. However, if the client is only interested in the current value of a business figure, e.g. a certain stock value, or if it is able to combine the values already received with the latest value on its own, the server system should choose an incremental update strategy, i.e. it will only send the delta changes and thus save network bandwidth and perhaps server memory as well. The combination of complex-structured context and the required functionality of delta propagation leads, for example, to the hybrid data model proposed within *PubScribe* (section 3.2).

3 Subscription Message Data Model

In this section, we briefly outline the underlying data model and the operators used to formulate standing queries. These operators are then subject of optimization and mappings to relational database systems, which will be shown in the subsequent section.

3.1 Types and Conditions of Subscriptions

From a theoretical point of view, a subscription may be represented as a mathematical function which is not yet saturated, i.e. the result of this function is still being computed or, in other words, the data which the computation of the function is based on is either not yet complete or changing over time. The bottom line for subscription systems from a database perspective is that a user registers a query once and regularly receives a notification of the query result derived from the actual state of the underlying data set. Therefore, the query may be considered the "body" of a subscription, which is subject to evaluation, if a corresponding delivery condition is met. Furthermore, subscriptions are instantiated, if corresponding opening conditions are satisfied. Analogously, subscriptions are removed from the system, if the present closing conditions evaluate to true.

Different Types of Subscriptions

The set of subscriptions can be classified into *feasible* and *non-* or *not yet feasible subscriptions*. A subscription on "the highest prime number twins" may be an example for a not-yet feasible subscription, because it is (still) unknown whether such numbers exist at all. Obviously, we have to restrict ourselves to feasible subscriptions. Moreover, we are able to classify these types of subscriptions in more detail from a data point of view into the following three categories:

- *Snapshot subscriptions*
  A snapshot [1] subscription may be answered by referring only to the currently valid information, i.e. the answer may be retrieved by processing only the most current message of a publisher. Snapshot subscriptions require "update-in-place" semantics.
  Example: A subscription regarding the *current* weather conditions only refers to the last available information. Old data is no longer of interest.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

- *Ex-nunc ('from now on') subscriptions*
  Ex-nunc subscriptions are based on a set of messages. This set of messages is constructed starting from an empty set at the time of the registration of a subscription.
  Example: Computing the value of a three-hour moving average of a stock price starts with a single value for the first hour, the average of two values for the second hour, and the average of three values for all other hours.

- *Ex-tunc ('starting in the past') subscriptions*
  Ex-tunc subscriptions are based on data from the past plus current information.
  Example: A subscription of the cumulative sum of trading information of a specific stock needs an initial overall sum, which can be maintained using new messages.

The *PubScribe* system supports (classic) snapshot-based, ex-nunc and ex-tunc subscriptions. To provide ex-tunc subscriptions, the system has to implement an initial evaluation mechanism, which provides feedback to the user on whether or not this specific subscription with the specified requirements can be instantiated.

Condition Evaluation Semantics

The evaluation of a subscription query (body of a subscription) is controlled by conditions. The *PubScribe* system uses the following three conditions to control the execution and delivery of a result of a subscription:

- *Opening condition*
  A subscription becomes active, i.e. the body and the following two conditions are instantiated as soon as the opening condition is satisfied the first time.

- *Closing condition*
  A subscription is removed from the system as soon as this condition evaluates to true.

- *Delivery condition*
  If and only if the delivery condition evaluates to true, the body of the subscription gets updated, i.e. messages which have arrived since the last delivery are "merged" into the current state of the subscription.

Once the opening condition is satisfied, the delivery and closing conditions are evaluated. If the delivery condition is satisfied, the subscription body is evaluated and the result is delivered to the corresponding subscriber.

If the closing condition evaluates to true, the subscription is removed from the system. It is worth to note here that the system provides "at least once" semantics for the execution of a subscription in the context of an initial evaluation for ex-tunc and one-shot subscriptions: the delivery condition is checked *before* the closing condition is evaluated. Thus, if the delivery condition is satisfied, the subscription body is evaluated and delivered before a satisfied closing condition removes the subscription from the system.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

header attributes | body attributes

StockName	StockExchange	Price	ChangeAbs	TradingVolume	TradingInfo
<StockName> Oracle </Stockname>	<StockExchange> FSE </StockExchange>	<Price> 97.50 </Price>	<ChangeAbs> 2.75 </ChangeAbs>	<TradingVolume> 3400 </TradingVolume>	<TradingInfo> <InfoSource> W. Lehner </InfoSource> <Comment> buy or die.... </Comment> </TradingInfo>

Fig. 5 Example for a MSGSET Data Structure

3.2 The *PubScribe* Message Data Structures

The data model of the *PubScribe* system consists of data structures and operators defined on these structures to formulate standing queries. The very interesting point within the hybrid modeling approach consists in the fact that the model reflects the duality of state-based and consumption-based data management by introducing message sequences and message sets.

Messages and Message Attributes

The messages produced by registered publishers must follow a message scheme announced during the registration process of a publisher at the notification management system. The scheme of a message $M = (H, B)$ consists of a header $H = (H_1,...,H_n)$, a (possibly empty) set of header attributes H_i ($1 \le i \le n$), and a message body $B = (B_1,...,B_m)$ with at least one body attribute B_j ($1 \le j \le m$). Header attributes may be seen as an equivalent to primary key attributes in a relational model [8] without the requirement of definiteness and minimality. The instances of attributes are valid XML documents [32, 17] and must follow an XML-schema [33, 34] definition, locally defined by the publisher. Attributes without further structuring are so-called basic or single-level attributes. Moreover, complex-structured attributes are not allowed in the header. Fig. 5 shows a single message regarding stock information. The complex attribute *TRADINGINFO* holds a comment together with the source of the quote.

Message Sets and Message Sequences

Messages of the same type may be collected as sets (unordered with regard to their generation time) or sequences [28]:

- *Message sequence (MSGSEQ)*
  The data structure of a message sequence holds a list of messages ordered by the arrival time of the message in the system. Each message in a sequence is implicitly extended by a header attribute *ValidTime*.
- *Set of sequences (MSGSET)*
  In order to reflect the stable set of information in addition to streaming data, MSG-

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

SET structures represent descriptive data to annotate incoming messages. From a logical point of view, a set of sequences reflects a consistent and (for that specific moment) complete state at every point in time.

Sample Scenario

Throughout the remainder of this contribution, a consistent example refers to a stock notification system about current trends, news, comments, and so on. A publisher StockInfo periodically delivers information about the current stock price added to a MEGSEQ structure. A second producer publishes comments on a fully incremental basis (section 2.3), i.e. the set of messages always reflects the current opinion of the publisher. Obviously, the messages go into a MSGSET structure. Fig. 5 shows an instance of the StockInfo publisher; Fig. 6 holds an example for a MSGSET regarding comments and rankings.

StockName	Ranking	Comment
<StockName> Oracle </Stockname>	<Ranking> **** </Ranking>	<Comment> Oracle is a member of the NASDAQ since ... </Comment>
<StockName> IBM </Stockname>	<Ranking> ***** </Ranking>	<Comment> IBM has a long tradition and .... </Comment>

Fig. 6 Example for a MSGSET Data Structure

3.3 The *PubScribe* Message Operators

The data structure may be used by operators to specify complex queries. Fig. 7 illustrates the data model and the underlying message operators to formulate a subscription. Within this query, only 5-star-ranked stocks after a join are considered. Based on the trading information, a dynamic window operation of size 3 is defined. Finally the average and the total volumes are computed as a result for the user.

The different operators are only sketched within this context. The reader is referred to [18] and [19] for a detailed description and a more comprehensive example:

- *Filter operator*
  The *FILTER()* operator is defined for header attributes. Hence, the resulting data structure holds only messages with values in the header attributes satisfying a given predicate. A selection criterion is restricted to conjunctive predicates without negation. Each predicate only contains the operators =, < , > and ~= for textual attributes. Example:
  [InterestedStocks] ⇐ *FILTER(StockName IN ('Oracle', 'IBM'))[StockInfo]*

- *Attribute migration operator*
  The attribute migration (*SHIFT()*-) operator allows the transition of a body attribute to the set of header attributes. The new header attribute must be of an atomic type. A good example for attribute migration is the definition of groups. For example, an

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

EVAL() operator extracts the month out of a time stamp stored in a body attribute. The *SHIFT()*-operator moves the newly created attribute to the set of header attributes providing a way to identify values on a monthly basis.

- *Internal message computation*
  The *EVAL()*-operator is used to perform computations within a single message. In fact, the model distinguishes three categories of internal message operators:

  The first category includes all regular binary scalar functions like *PLUS()*, *MINUS(), MULT()*, *DIV()* and equality operators (*GREATER()*, ...). Additionally, the class comprises a set of calendar functions like *YEAR()*, *MONTH()*, *DAY()*. The following example returns the relative change based on the current price, the price difference, and the turnover.

  [ExtendedStockInfo] ⇐ EVAL(Price, TradingVolume,
  ChangeRel:DIV(ChangeAbs, MINUS(Price, ChangeAbs)),
  Turnover:MULT(Price, TradingVolume))[StockInfo]

  The second category holds aggregation functions like *MIN()*, *MAX()*, *SUM()*, and *COUNT()* which are usually used in combination with the *COLLAPSE()* operator (see below).

  The third category encompasses all operators used to work on the content of complex-structured attributes. *EXTRACT()* is used to extract pieces of complex-structured attribute values. *COMBINE()* does the opposite: it merges two complex-structured attribute values to a new attribute value.

  [ExtractedStockInfo] ⇐
  EVAL(CommentList:EXTRACT(Comment, TradingInfoList),
  InfoSourceList:EXTRACT(InfoSource, TradingInfoList))[StockInfo]

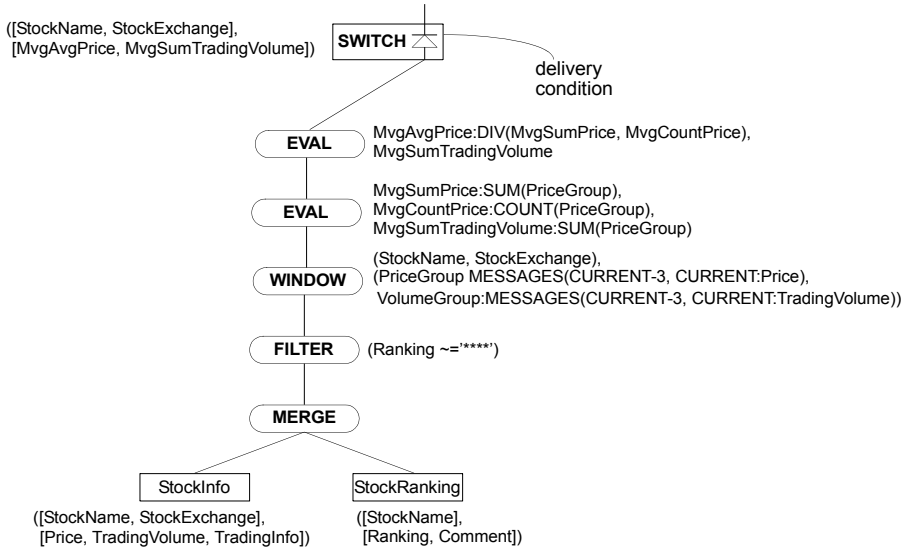


Fig. 7 Example for a Subscription Body

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

The identity function can be seen as a special form of one of these two operators. The syntax is abbreviated by just listing the attribute names.

- *Generation of static groups*
  The *COLLAPSE()* operator allows the definition of static (or: partitioning) groups with regard to a set of header attributes of a MSGSET or MSGSEQ structure. The following example groups message entries by the name of the stock value:

  GroupedStockInfo ⇐ COLLAPSE((StockName),
  (TradingVolumeGroup:TradingVolume,
  TradingInfoGroup:TradingInfo))[StockInfo]

  The result of a *COLLAPSE()* operator consists of head attributes (first parameter) and set body attributes (second parameter). A succeeding aggregation step has to be done explicitly by calling an *EVAL()* operator. The following expression computes the total sum as well as the number of contributing messages based on the attribute *TradingVolumeGroup*. Other attributes are not affected by this operation.

  SumStockInfo ⇐ EVAL(TradingVolumeSum:SUM(TradingVolumeGroup),
  TradingVolumeCount:COUNT(TradingVolumeGroup),
  TradingInfoGroup)[GroupedStockInfo]

  Furthermore, in contrast to the following dynamic group generation, the static groups are defined without regard to time or any other ordering characteristics.

- *Generation of dynamic groups*
  The main idea of dynamic grouping is that entities (i.e. messages) are grouped by a certain order and not by specific values. This implies that the definition of the *WINDOW()* operator is only based on MSGSEQ structures. The group definition can be performed either according to the number of messages or according to a time interval. The following example illustrates the effect of the operator, by defining an open window ranging from the first to the current entry and a sliding window covering all entries within a symmetrical 90-minutes slot:

  [WindowedStockInfo] ⇐ WINDOW((StockName),
  (TrVolOpenWindow:
  MESSAGES(BEGIN:CURRENT, TradingVolume),
  TrVolClosedWindows:
  TIMESTAMPS(-45.00:+45.00, TradingVolume)))
  [StockInfo]

  Analogously to the principle of static groups, dynamic groups have to be evaluated using an additional *EVAL()* operator, e.g.:

  [MvgSumStockInfo] ⇐ EVAL(TrVolCumSum:SUM(TrVolOpenWindow),
  TrVolMvgSum:SUM(TrVolClosedWindow))
  [WindowedStockInfo]

- *Merge operator*
  A merge operator joins two data structures of potentially different type and forms a new structure. Because this operator is crucial for the creation of more sophisticated results, it is described explicitly below.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

- *Switch operator*
  The special switch operator returns NULL as long as the control input (right input in Fig. 10) representing the condition testing is FALSE. Otherwise, the switch operator returns the messages from the data input. The *SWITCH()* operator is used to implement the test of various conditions.

Merging Two Message Data Structures

Within the *PubScribe* data model, a merge approach is applied relying on positions and values. Content-based joins are possible whenever a MSGSET is involved. A positional join is used to merge two MSGSEQ structures with multiple different join semantics. The non-commutative *MERGE()* operator implies four combinations as outlined below:

- *Join of messages in MSGSET structures*
  This case is comparable to a natural join in the relational model. More important, however, is the distinction between symmetric and asymmetric joins. In the first case, both messages exhibit the same set of header attributes. In the second case, one partner holds a superset of header attributes. If $H_1 \not\subset H_2$, $H_2 \not\subset H_1$ with $H_1 \cap H_2 \neq \emptyset$ holds, then the join is not defined; otherwise:
  $SET(H_1, B_1 \cup B_2) \Leftarrow SET_1(H_1, B_1) \bowtie SET_2(H_2, B_2)$
  with the join condition:
  $\forall(h_2 \in H_2)\ \exists(h_1 \in H_1)\ h_1 = h_2$

- *Join of messages in MSGSEQ with messages of MSGSET structures*
  This cross-structural join reflects the most important join in notification systems; incoming messages are enriched with additional information coming from relational sources using outer join semantics. If the set of header attributes in the MSGSET structure H_2 is not a subset of the header attributes of the MSGSEQ structure H_1, the join is not defined; otherwise:
  $SEQ(H_1 \cup \{ValidTime\}, B_1 \cup B_2) \Leftarrow SEQ(H_1 \cup \{ValidTime\}, B_1) \rtimes SET(H_2, B_2)$
  with the join condition independent of the time stamp attribute:
  $\forall(h_2 \in H_2)\ \exists(h_1 \in H_1)\ h_1 = h_2$

- *Join of messages in MSGSET with messages of MSGSET structures*
  Joins between messages from sets enriched with messages from sequences are not defined.

- *Join of messages in MSGSEQ structures*
  In addition to a join between messages of MSGSET structures, a positional join with $H_2 \subseteq H_1$ is defined as follows:
  $SEQ(H_1 \cup \{MAX(ValidTime_1, ValidTime_2)\}, B_1 \cup B_2) \Leftarrow$
  $SEQ(H_1 \cup \{ValidTime_1\}, B_1) \rtimes SEQ(H_2 \cup \{ValidTime_2\}, B_2)$
  The new message has the same valid time as the younger join partner. The join condition may be denoted as
  $\forall(h_2 \in H_2)\ \exists(h_1 \in H_1)\ h_1 = h_2$ and $\Theta(\text{ValidTime}_1, \text{ValidTime}_2)$.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

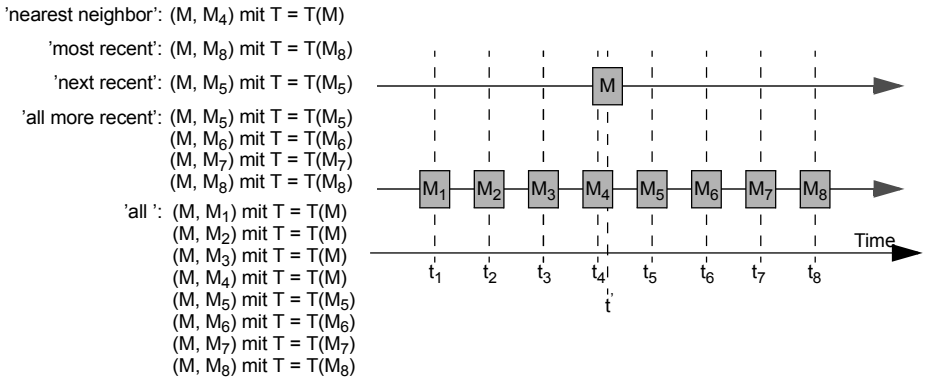


Fig. 8 Different Semantics for SEQ-SEQ Joins

The Θ-operator implies a huge variety of possible bindings. Fig. 8 shows multiple situations to find a join partner for a specific message *M*. A pointwise join features the following variations:

- *nearest neighbor*: candidate with the minimal distance to the message (M_4 in Fig. 8)
- *next recent*: candidate with the minimal timely forward distance (M_5)
- *most recent*: candidate with the highest time stamp values (M_8)

Additionally, interval joins are defined within the *PubScribe* data model to combine a single message with potentially multiple messages coming from the partner sequence:

- *all more recent*: set of messages with a valid time equal to or younger than the one of the reference message. In Fig. 8, message *M* would be combined with messages M_5 to M_8.
- *all*: a resulting message is produced for all members of the candidate sequence.

The following example shows a join between a MSGSEQ and a MSGSET; current stock prices, etc. are complemented by comments and rankings coming from a different source.

RankedStockInfo ⇐ *MERGE()[EVAL(Price, TradingVolume)[StockInfo],*
EVAL(Ranking)[StockRanking]]

The set of operators provides a solid base for a user to specify very complex descriptions (see table in the appendix for an overview). The notification system has to accept all feasible subscriptions and perform optimizations based on these structures. The processing model is outlined in the following section.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

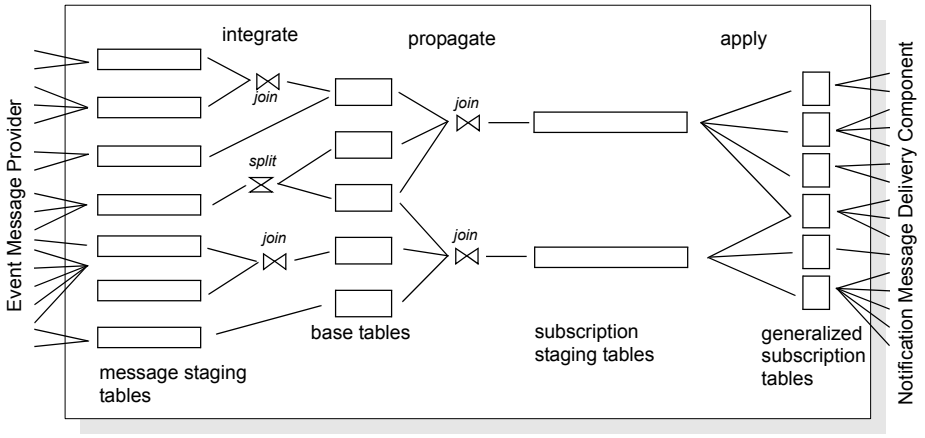


Fig. 9 *PubScribe* Message Processing Model

4 Subscription Processing Model

Using the set of operators and the structure, this section outlines the processing model of the *PubScribe* notification system, subdivided into a structural and an operational part.

4.1 Structural Layout and Processing Phases

The overall goal of the proposed *PubScribe* approach is to clearly decouple incoming messages from the resulting notifications as much as possible, and thus, to enable the notification system to optimize the processing of the subscriptions by operator clustering [27] and materialization [13]. As an underlying storage (!) model, *PubScribe* uses the relational model [8] and maps each message to a single row in a table. The system comprises multiple processing stages, each using different sets of tables as outlined below (Fig. 9):

- *Integration phase*
  Event messages are stored in message staging tables, where they are kept in their original (received) form. In a preliminary step, single messages may be integrated into base tables via join or split. Several options are possible: a message contributes to exactly one single base table; a message needs a join partner to generate an entry for a base table; or a message feeds two or more base tables, i.e. the content of a message is split into multiple entries in the base tables.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

The generation of notification messages is subdivided into three phases, which are introduced to share as much work as possible. For the coordination of this huge data pipeline, the system additionally introduces two sets of temporary tables:

- *Subscription staging tables*
  The purpose of staging tables is to keep track of all changes to the base tables, which are not yet completely considered by all dependent subscriptions. It is worth to mention here that a system may have multiple staging tables, and each staging table covers only these subscriptions which do only exhibit a lossless join. Lossy joins are delayed to the propagate or apply phase.

- *Generalized subscription tables*
  A generalized subscription table serves as basis for the computations of the notifications for multiple subscriptions (i) referring to the same set of base tables (at least a portion of them) and (ii) exhibiting similar delivery constraints. Each subscription may either be directly answered from a generalized subscription table, or retrieved from the generalized subscription table with either a join to another generalized subscription table or a back-join to the original base table. It is worth to note here that it must be ensured that the state of the base tables is the same as the state at the propagation time of the currently considered message.

It is important to understand that subscription staging tables are organized from a data perspective, whereas the set of generalized subscription tables is organized according to delivery constraints, thus providing the borderline from incoming to outgoing data.

- *Propagation phase*
  Comparable to the context of incremental maintenance of materialized views [13], the *PubScribe* system exhibits a second phase of propagating the changes from base tables to a temporary staging area. The resulting data is already aligned with the schema of the outgoing message, i.e. the relational peers of message operators (joins, selections, projections, and aggregation operations) have already been applied to the delta information. We have to mention here that the propagation appears immediately after the update of the base table.

- *Apply phase*
  The staging table holds accrued delta information from multiple updates. This sequence of delta information is collapsed, implying that the primary key condition is satisfied again and the resulting data is applied to one or more generalized subscription tables. In this phase, subscriptions exhibiting lossy joins are combined from entries of multiple staging tables or a back-join to the base tables. The result of the apply phase is picked up by the notification message delivery component and propagated to the subscriber.

Subdividing the process of subscription evaluation into multiple independent phases implies that the system has a huge potential for optimization. The basic strategies and the mapping to a relational query language are demonstrated below.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

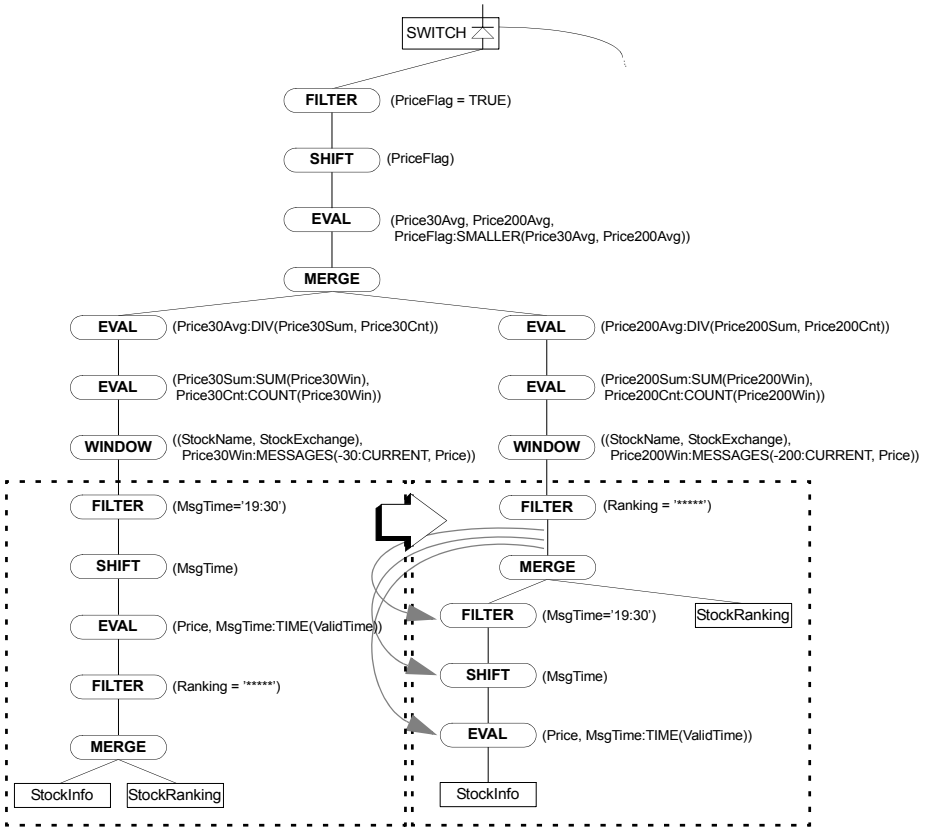


Fig. 10 Local Restructuring of *PubScribe* Subscriptions

4.2 Subscription Optimization and Relational Mapping

The optimization of subscription evaluation during the compilation is again subdivided into two phases. The first phase of local restructurings aims at the generation of a better execution plan using mechanisms restricted to the individual subscription. In this phase, the basic idea of optimizing relational queries is transferred to the subscription data model. A partially more important goal of this phase consists of generating a normal form, which reflects the working platform for the following inter-subscription optimization process. Fig. 10 shows a subscription plan with the same sub-expression before (left branch) and after (right branch) the local restructuring. The local operators *FILTER()*, *EVAL()*, and *SHIFT()* are pushed down to the leaf nodes.

The global subscription restructuring phase (second phase) targets the identification and exploitation of common sub-expressions by merging a newly registered subscription into an existing subscription network (first ideas published as Rete network in [11]). The merging process relies on the concept of building compensations. For exam-

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

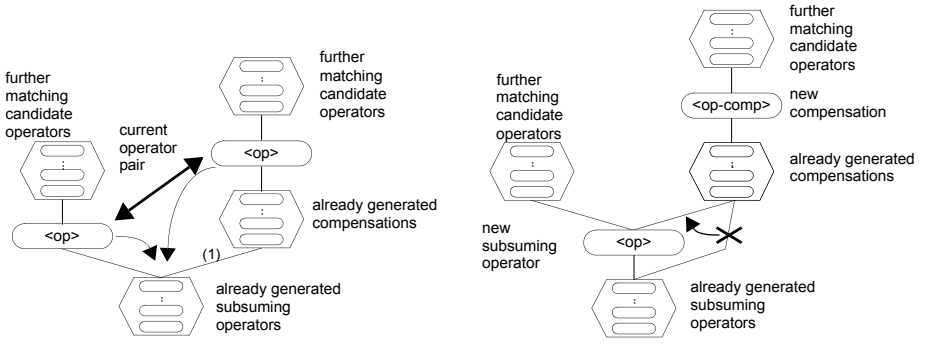


Fig. 11 Propagation of Multiple Operators

ple, if two expressions are of the same type and exhibit similar parameters, it might be worthwhile to compute the most general form of the operator only once and add compensation operators on top of the general form to produce the specific result for each consuming operator. Fig. 11 illustrates the process of using the stacks of operators and merging them step-by-step. As soon as a single pair matches at a specific level, the general form of the new subsuming operator and the two compensation operators for the individual subscription query are created. Additionally, the already generated compensations (with the newly created compensation operator and the operaters still to be matched on top of it) are now provided with data from the newly created subsuming operator. Obviously, if the operator does not produce all messages required by the lowest operator of the compensation, the whole matching procedure for that specific level fails. The overall process starts bottom-up and continues as far as possible (ideally up to the highest operator). The more similar the subscriptions are, the more operators can be shared. To enable pairwise comparison, the general form of the subscription—produced in the local restructuring phase—is extremely important.

The general process at a relational level is illustrated in [36, 20]. From the subscription-specific standpoint, it is worth to consider each operator regarding the matchability characteristic and the necessary compensations. Fig. 12 shows the most important operators and their corresponding compensations. A *FILTER()* operator regarding a predicate P_1 can be replaced by another *FILTER()* operator with a weaker predicate P_2 and a compensation consisting again of a *FILTER()* operator with the original or a reduced predicate to achieve the same result.

For a *SHIFT()* operator, the subsuming operator has to move only a subset of the attributes required by the matching candidate, such that the compensation moves the attributes still missing to the header of the message. The *EVAL()* operator can be easily compensated, if the subsuming operator generates all attributes required for the compensation, which introduces the scalar operations. In the case of a *COLLAPSE()* operator, a match is only successful, if the group-by attributes exhibit a subset relationship, i.e. the subsuming operator generates data at a finer granularity; the final groups can then be generated within the compensation. This first step is accomplished by a *COLLAPSE()* operator; the alignment of the grouping values requires an additional

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

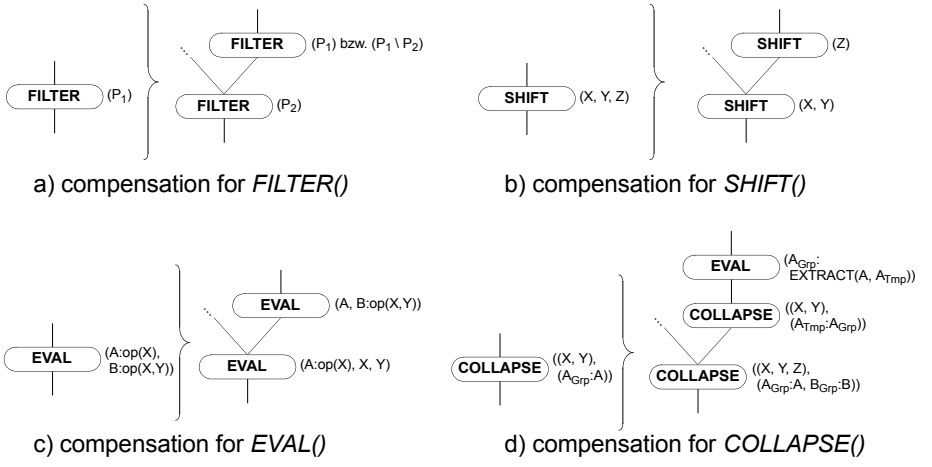


Fig. 12 Rules for Generating Compensations

EVAL::EXTRACT() operator to compensate for the additional nesting within the aggregation attributes.

For dynamic grouping, there is no easy way of building compensations [20]. A match is only successfully recorded, if the two parameter sets are equal, i.e. show the same window size and window characteristics. For *MERGE()* operators, a compensation depends on the similarity of the join predicates and the join type, i.e. the type (MSGSEQ or MSGSET) of the join partners. To weaken these restrictions, the join characteristics in the case of a MSGSEQ/MSGSEQ join can be exploited using the following partial ordering of the join characteristics:

$$ALL \prec (NEXT\ NEIGHBOR \mid ALL\ NEXT\ RECENT \prec (MOST\ RECENT \mid NEXT\ RECENT))$$

For example, a join with NEXT RECENT can be derived from a *MERGE()* operator with join characteristic ALL NEXT RECENT or simply ALL.

After restructuring the subscription query network both locally and globally, the final step consists in generating SQL expressions allowing the efficient mapping of the operator network onto operators of the relational storage model. Fig. 13 also shows this step for the branch next to the StockInfo publisher. In a last and final step, the database objects in a relational system are subject of pre-computation, if the corresponding operator in the subscription network has a potentially high number of consumers.

Fig. 13 illustrates the optimization process of a subscription operator network using our current example of stock trading information. It can be seen that the lowest two blocks (blocks denote relational database objects of either virtual or materialized views) of the operator network are the result of the matching process and reflect the set of subsuming candidates. The two parallel blocks denote compensations built on top of the commonly used query graph.

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

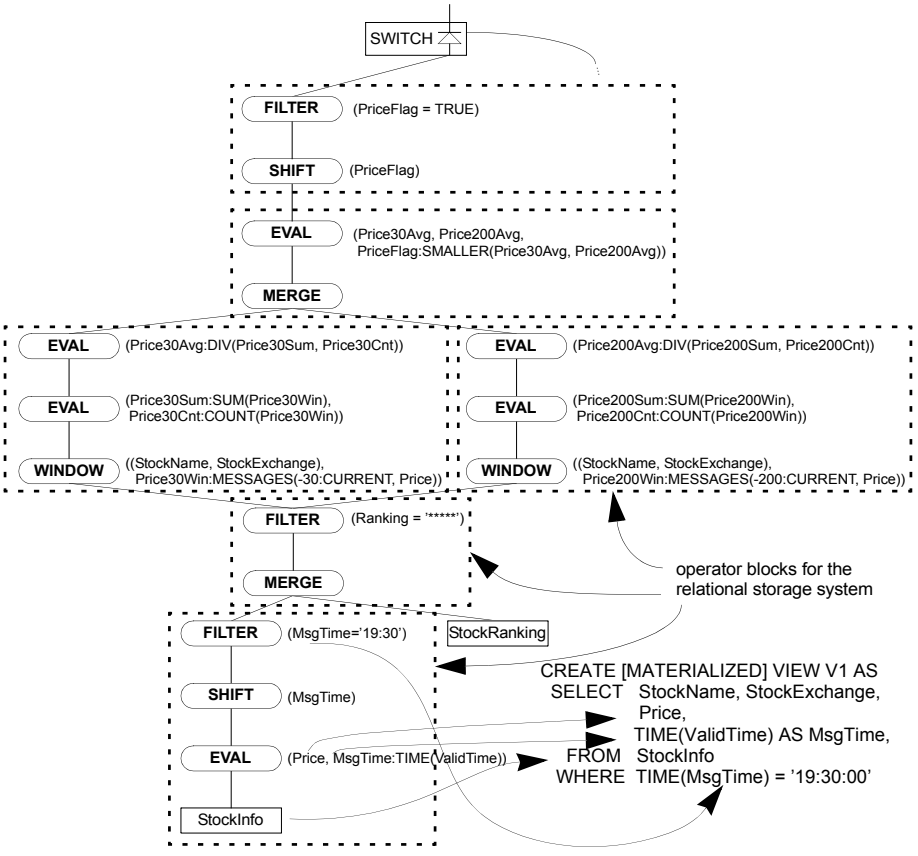


Fig. 13 Generating Subscription Execution Networks

5 Summary and Conclusion

Notification systems reflect a special kind of data management systems working like a huge data pipeline. Data items (documents) are entering the system, posted in form of event messages. Within the system, standing queries (subscriptions) are forming a complex-structured network of specific operators. Messages are routed through the operator network in multiple phases and finally arrive as notification messages at the delivery component responsible for sending out the messages in any supported format using a huge variety of protocols. In order to make these data pipelines work very efficiently and support a huge number of standing queries with similar structure, advanced database technology has to be adopted and exploited to a large extent. From a more global perspective with a database management system as an information provider, it is safe to

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

say that notification systems help the user to efficiently filter the vast amount of available information to focus only on relevant pieces of information.

References

[1] Adiba, M., Lindsay, B.: Database Snapshots. In: *Proceedings of the VLDB Conference, 1980*, pp. 86-91

[2] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: STREAM: The Stanford Stream Data Manager. In: *Proceedings of the SIGMOD Conference, 2003*, p. 665

[3] Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. In: *Proceedings of the SIGMOD Conference, 2000*, pp. 261-272

[4] Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: *Proceedings of the SIGMOD Conference 2002*, pp. 49-60

[5] Birman, K.P. : The Process Group Approach to Reliable Distributed Computing. In: *Communications of the ACM, 36(12), 1993*, pp. 36-53

[6] Bonnet, P., Gehrke, J., Seshadri, P.: Towards Sensor Database Systems. In: *Proceedings of the Mobile Data Management Conference, 2001*, pp. 3-14

[7] Carney, D, Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams - A New Class of Data Management Applications. In: *Proceedings of VLDB Conference, 2002*, pp. 215-226

[8] Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. In: *Communications of the ACM, 13(6), 1970*, pp. 377-387

[9] Cortes, C., Fisher, K., Pregibon, D., Rogers, A., Smith, F.: Hancock: A Language for Extracting Signatures from Data Streams. In: *Proceedings of the Knowledge Discovery and Data Mining Conference, 2000*, pp. 9-17

[10] Foltz, P.W., Dumais, S.T.: Personalized Information Delivery: An Analysis of Information Filtering Methods. In: *Communications of the ACM, 35(1992)12*, pp. 51-60

[11] Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In: *Artifical Intelligence, 19(1982)1*, pp. 17-37

[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997

[13] Gupta, A., Mumick, I.: *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999

[14] Hellerstein, J.M., Franklin, M.J, Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, B., Shah, M.A.: Adaptive Query Processing: Technology in Evolution. In: *IEEE Data Engineering Bulletin 23(2), 2000*, pp.7-18

[15] Koudas, K., Srivastava, D.: Data Stream Query Processing: A Tutorial. In: P*roceedings of VLDB Conference, 2003*, p. 1149

[16] Lehner, W.: *Datenbanktechnologie für Data-Warehouse-Systeme* (in German). dpunkt.verlag, 2003

[17] Lehner, W., Schöning, H.: *XQuery: Grundlagen und fortgeschrittene Methoden* (in German). dpunkt.verlag, 2004

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

[18] Lehner, W.: *Marktplatz omnipräsenter Informationen - Aufbau und Analyse von Subskriptionssystemen* (in German). B.G. Teubner Verlag, 2002

[19] Lehner, W. (Hrsg.): *Advanced Techniques in Personalized Information Delivery*. Technical Report, University of Erlangen-Nuremberg, 34(5), 2001

[20] Lehner, W., Pirahesh, H., Cochrane, R., Zaharioudakis, M.: fAST Refresh using Mass Query Optimization. In: *Proceedings of the ICDE Conference, 2001*, pp. 391-398

[21] McCarthy, D.R., Dayal, U.: The Architecture Of An Active Data Base Management System. In: *Proceedings of the SIGMOD Conference, 1989*, pp. 215-224

[22] Oki, B.M., Pflügl, M., Siegel, A., Skeen, D.: The Information Bus - An Architecture for Extensible Distributed Systems. In: *Proceedings of the SOSP Conference,1993*, pp. 58-68

[23] Powell, D.: Group Communication (Introduction to the Special Section). In: *Communications of the ACM 39(4), 1996*, pp. 50-53

[24] Pu, Y., Liu, L.: Update Monitoring: The CQ Project. In: *Proceedings of the International Conference on Worldwide Computing and Its Applications, 1998*, pp. 396-411

[25] Ramakrishnan, S., Dayal, V.: The PointCast Network. In: *Proceedings of the SIGMOD Conference, 1998*, pp. 520

[26] Roth, M.T., Schwarz, P.M.: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In: *Proceedings of the VLDB Conference, 1997*, pp. 266-275

[27] Sellis, T.: Multiple Query Optimization. In: *ACM Transactions on Database Systems, 13(1), 1988*, pp. 23-52

[28] Seshadri, P., Livny, M., Ramakrishnan, R.: Sequence Query Processing. In: *Proceedings of the SIGMOD Conference, 1994*, pp. 430-441

[29] Sullivan, M., Heybey, A.: Tribeca: A system for managing large databases of network traffic. In: *Proceedings of the USENIX Annual Technical Conference, 1998*

[30] Terry, D.B., Goldberg, D., Nichols, D., Oki, B.M.: Continuous Queries over Append-Only Databases. In: *Proceedings of the SIGMOD Conference, 1992*, pp. 321-330

[31] Tian, F., Reinwald, B., Pirahesh, H., Mayr, T., Myllymaki, J: Implementing a Scalable XML Publish/Subscribe System Using a Relational Database System. In: *Proceedings of the SIGMOD Conference, 2004*, pp. 479-490

[32] World Wide Web Consortium: *Extensible Markup Language (XML), Version 1.0, Second Edition*. W3C Recommendation.
Electronically available at: `http://www.w3.org/TR/2000/REC-xml-20001006`

[33] World Wide Web Consortium: *XML Schema Part 1: Structures*.
Electronically available at: `http://www.w3.org/TR/xmlschema-1/`

[34] World Wide Web Consortium: *XML Schema Part 2: Datatype*s.
Electronically available at: `http://www.w3.org/TR/xmlschema-2/`

[35] Yan, T.W., Garcia-Molina, H.: SIFT - a Tool for Wide-Area Information Dissemination. In: *Proceedings of the USENIX Winter Conference, 1995*, pp. 177-186

[36] Zaharioudakis, M., Cochrane, R., Pirahesh, H., Lapis, G., Urata, M.: Answering Complex SQL Queries Using Summary Tables. In: *Proceedings of the SIGMOD Conference, 2000*, pp. 105-116

Final edited form was published in Theo Härder, Wolfgang Lehner, Hgg., 2005. "Data Management in a Connected World." Berlin: Springer, S. 111–136. ISBN 978-3-540-31654-1.
https://doi.org/10.1007/11499923_7

Appendix

Tab. 1 Listing of all *PubScribe* Operators

Description	Operator Specification
filtering	X' ⇐ FILTER(<attr> [= , ~=, <, >] <val>, <attr> IN (<val_1>, ..., <val_n>), ...)[X]
attribute migration	X' ⇐ SHIFT(<attr>, ...)[X]
attibute operator - scalar operator - aggregations operator - structural modification operator	X' ⇐ EVAL(<attr>, <attr'>:<scalar-op>(<$attr_1$>[,<$attr_2$>]), <attr'>:<aggr-op>(<attr>) <attr'>:<attr-op>(<$attr_1$>, <$attr_2$>), ...)[X] <scalar-op> ∈ { PLUS, MINUS, MULT, DIV } <scalar-o1opd> ∈ { GREATER, SMALLER, EQUAL } <scaler-op> ∈ { DATE, YEAR, MONTH, DAY } <scaler-op> ∈ { TIME, HOUR, MIN } <aggr-op> ∈ { MIN, MAX, SUM, COUNT } <attr-op> ∈ { EXTRACT, COMBINE }
static group by	X' ⇐ COLLAPSE((<$attr_1$>, ..., <$attr_n$>) (<$attrGrp_1$>:<$attr_1$>), ..., (<$attrGrp_m$>:<$attr_m$>))[X]
dynamic group by	X' ⇐ WINDOW((<$attr_1$>, ..., <$attr_n$>), (<$attrWin_1$>:<win-spec>(<start>:<stop>, <$attr_1$>), ..., <$attrWin_m$>:<win-spec>(<start>:<stop>, <$attr_m$>))[X] <win-spec> ∈ { MESSAGES, TIMESTAMPS } <start> ∈ { BEGIN, CURRENT, <int-val>, <time-val>} <stop> ∈ { END, CURRENT, <int-val>, <time-val>}
join	X' ⇐ MERGE(<join-spec>)[X_1, X_2] <join-spec> ∈ { NEXT NEIGHBOR, MOST RECENT, NEXT RECENT, ALL RECENT, ALL }