CISC 499*: Transactions and Data Stream Processing

Neil Conway (471-0246)

April 6, 2008

Abstract

The current generation of data stream management systems (DSMS) do not provide transactions. This is unfortunate: in this paper, we argue that transactions are a relevant concept for data stream processing, because they simplify the development of correct, reliable applications. As a first step toward a complete transaction model for stream processors, we present several situations in which existing DSMS systems are inadequate, and describe how transaction-like behavior provides a clean solution. We conclude by discussing what these examples suggest about the proper role for transactions in a DSMS.

1 Introduction

In database systems, correctness criteria are usually defined in terms of *transaction serializability*: database operations are grouped into atomic transactions, and the system guarantees that an application's transactions will be executed in a manner that is equivalent to some serial ordering. Combined with the other ACID properties, transactions simplify the development of database applications[GR93].

Unfortunately, there is no equivalent set of concepts for the current generation of data stream systems. DSMS do not generally provide transactions, and typically only make informal guarantees of correctness. In fact, it is not even clear how the concept of a transaction ought to be applied to data stream processors: because queries are typically read-only and specified in a declarative query language, the evaluation of a continuous query usually does not interfere with other queries. The DSMS has few constraints on the order in which it evaluates queries, and the typical guarantees of transaction serializability made by database isolation models are not relevant.

Nevertheless, we believe that the transaction concept can be usefully applied to data stream systems. Rather than focusing on serializability, transactions in a DSMS are *data-oriented*: they provide guarantees about the movement of data into, within, and out of the DSMS. We associate transaction boundaries in a database with *window boundaries* in a data stream: a data stream's window is proposed as the basic unit for data flow in the DSMS. For example, windows are used as the unit of isolation for continuous queries, and the unit of durability for archived streams, and the output streams of the continuous queries themselves. In this paper, we describe several situations in which this “transaction-like” behavior would improve the behavior of a DSMS, and then sketch an architecture for a transaction-oriented DSMS.

The remainder of this paper is organized as follows. We begin by introducing background material, and discussing relevant prior work. In Section 2, we summarize the concept of a transaction, and argue for why transactions have proven to be a useful way to structure database applications. In Section 3, we describe the components of a typical data stream management system, and highlight some of the differences between data stream and database systems that are particularly relevant to transactions. Section 4 contains motivating examples of how transactional behavior would be useful in a DSMS for isolation (4.1), durability (4.2), and crash recovery (4.3). In Section 5, we analyze what these examples have in common, and discuss what they suggest about a potential architecture for a transaction-oriented DSMS.

2 Transactions

A *transaction* is a sequence of related database operations that are treated as an atomic unit. [Gra81] traces the transaction concept to contract law: two or more parties negotiate until they have agreed upon a set of state transformations. Once an agreement has been reached, the contract is finalized by a signature or another symbol of joint commitment. It is only when this committal has been made that the changes described by the transaction are binding.

In this section, we begin by defining the basic properties of the transaction concept (Section 2.1). We then discuss extensions of the basic transaction concept to account for distributed systems and reactive applications (Section 2.2). We summarize the traditional DBMS approach to transaction isolation in Section 2.3), and other related prior work (Section 2.4).

2.1 The ACID Properties

Database systems guarantee that transactions possess the so-called "ACID" properties[GR93]:

atomic: Either all the database operations in a transaction are completed, or none of them are.

consistent: A transaction must not leave the database in an inconsistent state. If each transaction preserves the consistency of the database when run in isolation, the database system ensures that the database remains consistent when transaction execution is interleaved.

isolated: A transaction is isolated from the effects of concurrent transactions. In the strictest interpretation, isolation requires that the system provide the illusion that each transaction is executed serially. In some systems this requirement is relaxed, and the system merely ensures that modifications made by other transactions are made visible to a transaction in a well-defined way.

durable: Once a transaction has been committed, the system guarantees that its effects are persistent, even in the face of a subsequent system crash.

By providing ACID semantics, database systems simplify client applications. For example, providing atomicity lifts the burden of considering intermediate states from application developers: the developer need only ensure that the transaction leaves the database in a consistent state if it is run to completion. Isolation simplifies reasoning about the way in which a transaction can interact with concurrent database operations. Similarly, if the system guarantees that all committed transaction are durable, the application developer typically does not need to concern themselves with recovering database state in the event of a system crash.

Another benefit of transactions is that they allow applications to be more easily *composed* from collections of related operations[HMPJH05]. This encourages modularity in design: each transaction can be developed and verified in relative isolation, and then combined with other transactions in a well-defined way.

2.2 Extended Transaction Models

The canonical use case for transactions involves moving money between customer accounts at a bank[GR93]. Transactions provide a natural way to structure applications in order to ensure that no money is lost or mistakenly allocated to more than one account. Many simple transaction-processing applications are conceptually similar to this example: they involve transactions that last for a relatively short time (*short-lived)*, and consist of simple state transformations that modify a single database (*centralized)*.

Certain applications require a more sophisticated transaction model. For example, consider a trip planning application that automates the process of booking a complicated journey[GR93]. Planning a trip requires interacting with a wide variety of data stores, including databases associated with airlines, car rental agencies, hotels, and travel agents. These systems may be geographically distributed, and likely do not belong to the same organizational or administrative domain. Therefore, the trip planning application would like to execute a *distributed transaction* that involves modifications to a collection of heterogeneous data stores. Each such modification should be transactional, so the operation is a whole can be modeled as a group of related *sub-transactions*. Decomposing the trip planning process into a set of related sub-transactions simplifies control flow and error handling. Other applications ill-suited to the simple transaction model include "open-ended" collaborative applications like CAD and software engineering[PK88], and bulk update operations[GR93].

Numerous "extended" transaction models have been proposed, including savepoints, nested transactions[Mos85], multi-level transactions[BSW88], split transactions[PK88], and sagas[GMS87]. The details of these various models are beyond the scope of this paper, but they all attempt to extend the transaction model to handle different kinds of long-running, distributed operations with complex internal control flow.

When the transaction model is extended in this way, maintaining the strict ACID properties is often challenging[GR93]. For example, providing the illusion of atomicity and serial execution is inherently more difficult when transactions are long-running and involve multiple computer systems. Transactions are often implemented using locks; as the duration of transactions increases, the chance of deadlock also rises[Gra81]. Broadly speaking, the situation can be improved by relaxing the ACID guarantees, or by dividing a long-running transaction into smaller atomic units, and then making guarantees about the relationships between those sub-units.

2.3 Isolation Models

An *isolation model* defines the possible behaviors that a transaction may witness during its execution. Since one database transaction can directly interfere with another, isolation models typically focus on defining the ways in which the system may interleave the execution of concurrent transactions. The standard approach is *serializability*: the database system is free to execute transactions in any order, provided that it chooses an execution order that is equivalent to some serial ordering of transactions[GR93]. Because implementing true serializability may be expensive and can result in aborting unserializable transactions, several more relaxed consistency criteria have been proposed (e.g. [GLPT76]). This allows the application developer to select the appropriate tradeoff between consistency and performance.

Isolation models are usually formalized in terms of constraints upon legal *transaction schedules* (execution histories): the model specifies the kinds of transaction schedules (execution interleavings) that do not violate the system's consistency constraints. A formal, mathematical isolation model forms the basis for the DBMS's concurrency control strategy[GR93].

Each of the extended transaction models discussed in Section 2.2 typically includes an isolation model that specifies the constraints that a correct implementation of the transaction model must satisfy. ACTA is an attempt to generalize these isolation models into a formal framework for specifying and reasoning about extended transaction models[CR94, BDG$^+$94].

2.4 Prior Work

Transactions have long been a component of information processing systems. [Gra81] provides a summary of the transaction concept and some of its common applications. The first transaction isolation model was also proposed by Gray[GLPT76], under the name "degrees of consistency." Gray's original proposal was implementation-dependent, in the sense that it required a lock-based implementation of concurrency control. Nevertheless, Gray's work on degrees of consistency was used as the basis for the "isolation level" concept introduced in the SQL-92 standard, which was intended to be implementation-independent. Subsequent work observed that the definition of isolation levels in SQL-92 specified ambiguous behavior in some circumstances[BBG$^+$95]. [ALO00] proposed an isolation model that was both unambiguous and implementation-independent.

The transaction concept has also been successfully applied outside the domain of databases and transaction processing. [HM93] introduced transactional memory as a way of constructing concurrent programs without using locks or traditional lock-free techniques. Although the initial formulation depended on hardware support for transactional memory, the idea was later generalized to software transactional memory (STM)[ST95]. STM has been the subject of much subsequent work in the programming language community. For example, [HMPJH05] presents an elegant approach to using software transactional memory to constructing concurrent Haskell programs.

The idea of implementing long-lived transactions as a group of smaller, atomic transaction-like units is also not new. [GMS87] proposes *sagas*, which are long-lived database operations that are composed of one or more transactions. Each of these subsidiary transactions is allowed to commit or abort independently. The saga as a whole can commit if each of its subsidiary transactions has committed successfully; otherwise, the entire saga must be aborted. Aborting a saga requires reversing the effects of any of its component transactions that have committed. This is done through the use of "compensating transactions" that restore the logical state of the database to how it was before the transaction was executed.

3 Data Stream Management Systems

Database management systems have been successfully applied in many applications and to many different domains. Classical DBMS systems use a "store and query" model: data is stored permanently in the database, and queries are evaluated against the database's current state and then discarded. While this architecture will continue to be useful, it has recently been recognized that this model is inappropriate for an emerging class of data management problems, in which applications want to perform real-time analysis over a set of infinite, continuous data streams. Data Stream Management Systems (DSMS) have been proposed to meet the challenges raised by these new applications[GÖ03]. Typical applications from this class include include network traffic monitoring[CJSS03, PGB$^+$04], real-time financial analysis[LS03], and sensor networks[BGS01]. In this section, we summarize the major components of a DSMS, and highlight some of the differences between databases and data streams.

A Data Stream Management System consists of the following components:

1. *data streams*, which represent the input to the system (Section 3.1)
2. *continuous queries*, which express conditions of interest over one or more data streams (Section 3.2)
3. *query processors*, which accept input from a set of data streams, evaluate a set of continuous queries, and produce a set of output data streams (Section 3.3)
4. *clients*, which submit continuous queries and process query results (Section 3.4)

The architecture of a typical DSMS is illustrated in Figure 1. In Sections 3.1–3.4, we describe each component of a DSMS in more detail. We highlight notable differences between database and data streams systems, and summarize those differences in Section 3.5.

3.1 Data Streams

A *data stream* is an infinite bag (multiset) of tuples, including a timestamp attribute[ABW03]. The timestamp attribute specifies the position of the tuple within the data stream. Note that in contrast to traditional relational systems, streams are inherently ordered.

There are two kinds of streams: *base streams* and *derived streams*[ABW03]. Base streams represent the external input to a data stream system, and are typically derived from a continuous data source in the physical world (e.g. network traffic, a sensor, or a "ticker" describing the changes in a financial market). The rate of arrival of a stream is typically unpredictable, and can often be "bursty": the stream's peak arrival rate far exceeds its average rate. Because stream tuples typically arrive faster than they can be written to disk, the DSMS must

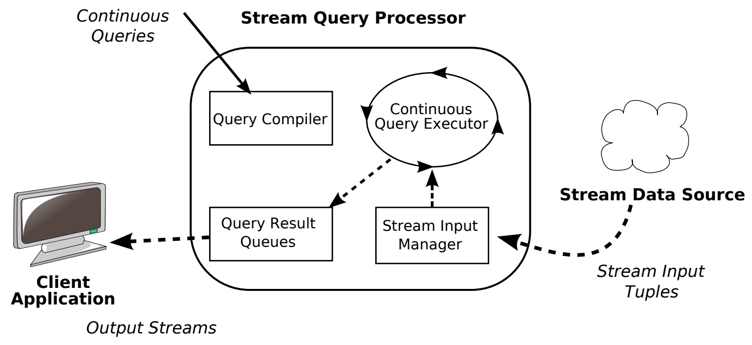

Figure 1: The architecture of a typical DSMS.

usually process them as they arrive and then discard them, or archive them to disk in the background[CCD+03].

Derived streams are intermediate streams that are produced by operators in the stream query language. They can be used to compose multiple data streams to simplify a complex query, in a manner analogous to how views are used in traditional database systems.

There are two kinds of stream timestamps: *explicit timestamps*, which are included in the data stream when it arrives at the DSMS, and *implicit timestamps*, which are assigned by the DSMS[SW04]. Explicit timestamps are typically derived from a physical clock, and usually describe the time at which an event occurred in the outside world, whereas implicit timestamps can be derived from a logical clock, and are typically based on the order in which tuples arrive at the DSMS. Explicit timestamps are more relevant to many real-world applications, but also introduce some additional complications: for example, externally-supplied timestamps may not be unique, and the stream tuples might arrive out-of-order[SW04]. Therefore, this paper assumes that streams are implicitly timestamped, so that the timestamp attribute obeys a total order that agrees with the arrival time of the input tuples.

Note that from the perspective of the DSMS, stream tuples do not exist until they have been produced by a base stream. Prior to this point, the data is not visible to the DSMS, and is therefore not subject to any correctness guarantees that the DSMS may provide (e.g. with regard to durability or visibility).

3.2 Continuous Queries

A *continuous query* is a query that accesses one or more data streams, and produces an unbounded stream of results. For the sake of comparison, we use the term *snapshot query* to refer to the “one-time” queries supported by traditional database systems.

In a DBMS, queries are typically expressed in a declarative query language, and (logically) applied to the entire database: the predicates in the query indicate which tuples satisfy the query. This approach cannot be directly applied to data stream systems: because streams are infinite, a traditional relational operator would never terminate when applied to a stream. Clearly, a different query language is needed to work with data streams.

There are several approaches to streaming query languages, including object-based languages[BGS01], graphical data-flow languages[CCC$^+$02], and relation-based languages[LS03, CCD$^+$03]. In this paper, we focus on relation-based query languages that are similar in principle to the Continuous Query Language (CQL) defined by the Stanford STREAM project[ABW03].

CQL embodies a conservative approach to language design. Relational query languages are well understood, and have proven to provide a sufficient set of constructs for manipulating relational data. CQL takes an existing relational query language (such as SQL) and extends it to handle streams, allowing CQL to benefit from prior work on relational query language semantics and optimization.

CQL extends a relational query language with some new types and operators for handling streams:

- a *stream* is a new type of database object, with the properties described in Section 3.1
- *stream-to-relation* operators periodically produce relations from finite sub-sets of a stream. Window operators are the primary stream-to-relation constructs, and are discussed further in Section 3.2.1
- *relation-to-stream* operators produce data streams from the content of a relation. As a relation's contents change over time, those changes can be emitted as a data stream. CQL defines three stream-to-relation operators:
    1. The RSTREAM of relation R at time t consists of the entire content of R at t.
    2. The ISTREAM of a relation R at time t is defined as $R_t - R_{t-1}$. That is, the ISTREAM contains all the tuples that were added to R between time $t-1$ and t.
    3. The DSTREAM of a relation R at time t is defined as $R_{t-1} - R_t$. The DSTREAM is the converse to ISTREAM: it contains the tuples that were recently removed from R.

In CQL, the result of a query is a *stream*, not a relation. The primary input to the system is the set of base data streams. Intuitively, it follows that a query in CQL is typically composed of three components: a stream-to-relation operator that selects finite sub-sets of data from one or more data streams to process, a set of relational operators that filter and transform this data, and finally a relation-to-stream operator that produces an output data stream from the result of the relational operators. Because each query is essentially a transformation from data streams to data streams, multiple queries can be composed through the use of derived streams.

Typically, if the query's relational component involves grouping and aggregation, RSTREAM is often the appropriate operator to apply to produce the output stream. Otherwise, ISTREAM is commonly used. Requiring the programmer to specify an explicit stream-to-relation operator leads to verbose queries. To allow simple queries to be expressed concisely, CQL defines a default stream-to-relation operator for some queries[ABW03]. For the sake of clarity, example queries in this paper will not take advantage of any syntactic defaults.

In a DBMS, a query accesses a single, consistent snapshot of the database. It seems unlikely that the same will be true of continuous queries. Requiring a continuous query to use a single, static snapshot of the database would be both expensive, because of the need for long-lived locks or transactions, and also undesirable, since most queries will want to incorporate updates to database contents at some point during their lifetimes. The problem of isolation and continuous queries is further explored in Section 4.1.

3.2.1 Windows

A query is only interested in a finite subset of a stream at any given point, although this subset usually changes over time. Furthermore, most applications are interested in the *most recent* content of a stream. These observations give rise to the concept of a *window operator*, which is the prototypical stream-to-relation operator.

In CQL, a sliding window is applied to a data stream and periodically produces a relation. The size of the window is specified by its *range*, and the window's period is specified by its *slide*. We use the term *window boundaries* to describe the points at which a window slides. Conceptually, at every window boundary, the DSMS produces a relation containing all the stream tuples within the window's range. Both the slide and range of a window can be specified in time-based or tuple-based units. For example, Algorithm 1 computes the total number of shares traded in the most recent 5 minutes, on a per-stock basis. Every 5 minutes, this query produces a new set of results by computing the query expression on the most recent 5 minutes of data in the stream.

Algorithm 1 A simple example query in CQL.

```
SELECT RSTREAM(t.symbol, sum(t.volume))
FROM ticker t
   [ SLIDE BY ‘5 minutes’ RANGE BY ‘5 minutes’ ]
GROUP BY t.symbol;
```

Inspired by [KWF06], we classify a window according to its range r and slide s as follows:

- If $r \leq s$, then the window is *disjoint*: each stream tuple appears in at most one window
- If $r > s$, then the window is *overlapping*

Stream tuples that do not fall within any windows are effectively ignored. Note that in overlapping windows, a single stream tuple can appear in two or more windows. Logically, each window is treated as a distinct set of tuples. In practice, the DSMS may optimize query execution by taking advantage of the commonalities between overlapping windows[KWF06].

The windows described above are properly known as *sliding windows*, since both endpoints of the window change with time. A window in which both endpoints are fixed is termed a *fixed window*, whereas a *landmark window* has a single fixed endpoint[GÖ03]. We focus on sliding windows for the remainder of this paper.

3.2.2 Mixed Joins

In a DSMS that uses a relation-oriented query language, it is natural for the system to provide both streams and relations, and to allow a single query to access both types of objects. In fact, this capability proves to be quite useful in practice, since applications frequently need to lookup mostly-static reference information that is associated with incoming stream tuples. For example, a system accepting a stream of equity market ticker updates might need to lookup additional information about the equities that is not present in the stream itself (historical information about a stock, for instance). This operation can be elegantly modeled as a join between a stream and a table, which we call a *mixed join.*[1] Algorithm 2 includes an example query that illustrates this concept.

Algorithm 2 An example query that uses a mixed join to compute the total value of trades in a certain basket of "interesting" stocks every 10 minutes.

```
SELECT RSTREAM(s.name, sum(t.volume * t.price))
FROM stocks_of_interest s,
     ticker t
   [ RANGE '10 minutes' SLIDE BY '10 minutes' ]
WHERE s.symbol = t.symbol
GROUP BY s.name;
```

A mixed join consists of a stream, a relation, a window operator, and an optional join predicate. Conceptually, the mixed join applies the window operator to the stream. Each window's worth of tuples is joined against the relation according to the join predicate (if any), and then emitted as the output of the mixed join. Note that, like a window operator, a mixed join produces an infinite sequence of finite relations, *not* a data stream.

This approach to implementing mixed joins is undesirable in some circumstances: buffering a window's worth of results in memory before computing the

[1]In several cases, previous work has made casual references to this concept: for example, [GÖ03] notes that it would be desirable for DSMS to allow joins between streams and "static meta-data". However, we are not aware of a previous in-depth treatment of this construct. The name "mixed join" is due to S. Krishnamurthy.

join might require too much memory if the window is large. In fact, in many cases there is no need to keep an entire window's worth of data in memory at any time. We can take advantage of the commutativity of the join predicate and the window operator to compute the results of a mixed join incrementally. For each incoming stream tuple, the DSMS can immediately join the tuple against the relation, and then subsequently apply the window operator to the resulting join tuples. This technique can be applied to the mixed join query given in Algorithm 2 to avoid buffering stream tuples in memory. We expect that most practical data stream systems will use this approach to evaluating mixed joins.

Note that the relation operand to a mixed join does not need to be a base relation: it may be the result of applying a query expression to one or more base relations. Similarly, the stream operand may be either a base stream or a derived stream. Therefore, a mixed join essentially represents a "crossover point" between two components of a query plan: a *passive* subplan that produces a relation, and an *active* subplan that produces a stream.

Many data stream systems also support *window joins*, which are joins between two or more data streams[KNV03]. Window joins are useful for correlating recent information from multiple data streams. As such, they complement mixed joins, which are typically useful for correlating recent stream data with historical or static information.

3.3 Stream Query Processors

A stream query processor takes a set of data streams as input, and evaluates the continuous queries in the system to produce a set of output data streams. Data streams are "injected" into the system by connecting to the query processor's input manager. The input manager makes stream tuples available to the query executor, which then periodically makes query results available to clients.

3.3.1 Query Evaluation

The difference between stream-oriented and relation-oriented query languages is more than superficial: data stream systems are amenable to completely different query optimization and evaluation strategies than traditional databases.

In a DBMS, data is long-lived, but queries are transient. The DBMS responds to each new query by evaluating the query relative to the current state of the database, returning a finite result set, and then discarding the query. In contrast, continuous queries in a DSMS are long-lived, but data is transient. Activity in a DSMS is usually initiated by new *data* (the arrival of incoming stream tuples), rather than by the arrival of new queries. Therefore, a stream query processor is naturally implemented by applying incoming stream tuples to a set of continuous queries, emitting new result data stream tuples as necessary, and then discarding the input stream tuples. Query execution for stream processors is often implemented by routing stream tuples through a graph of operators.

A single stream processor may execute many concurrent continuous queries over the same set of data streams, so sharing evaluation work among similar queries is essential for good performance[CCD$^+$03]. In fact, the long-lived nature of most continuous queries makes them more amenable to shared processing. However, it is worth noting that sharing the evaluation work between two queries may have subtle implications on the snapshot of system state seen by those queries.

In a stream query processor, most of the important state information in a DSMS is resident in memory: in state associated with the nodes of the graph of operators, for example. Some DSMS may not even have an attached disk, and many stream processors will not write to disk frequently during the course of their operation. In contrast, the important state in a database system is disk resident. The DBMS ensures that data is persisted to stable storage, and most database systems are content to abort in-progress transactions in the event of a system crash. As long as the on-disk state in a database system can be made consistent, the system can recover from crashes successfully.

3.4 Clients

In a database system, the client model is simple: clients submit queries to the database system and then block until the results are available. Queries are typically embedded directly in the client application, either as textual strings in the application's source code, or via techniques like SQL/CLI.

Relatively little attention has been paid to the role of the client in DSMS systems, but we observe that there are fundamental differences between stream clients and database clients:

- Continuous queries produce a stream of results, rather than a relation. Therefore, a client typically does not (and cannot) consume the entire output stream at once; instead, the DSMS emits an output stream of results, which is periodically consumed by the client

- Because continuous queries can commonly last for weeks or months, it would be awkward for client applications to remain connected to the DSMS for the duration of query execution

- Clients are often latency-sensitive: many DSMS clients would like to be notified of new query results as soon as possible. Therefore, requiring clients to poll for new results is inconvenient: many applications will either need to poll the DSMS frequently, or incur additional latency in result processing

It follows that while a "pull"-style interface is convenient for database clients, a "push"-style interface to query results is more appropriate for many DSMS applications. This suggests that we should consider the DSMS to be only one component in a sequence of systems that perform actions in response to incoming stream events. Rather than delivering query results directly to client applications, it is often preferable to have the DSMS push results to a middleware component (such as a message queue[Gra95]), which can then make the results available to clients as necessary. Such "message-oriented middleware" (MOM) systems often provide a "publish/subscribe" interface that can promptly notify clients when new results are available[EFGK03]. Using a MOM system to store query results encourages *loose coupling* between the DSMS and the client: the client and the query processor do not need to be online at the same time, for example. This is a useful property in the context of long-lived, continuous queries.

An interesting observation is that middleware systems are similar to traditional database systems in various ways, including support for transactions[Gra95]. If the data stream processor were to also provide a notion of transactions, two-phase commit could be used to ensure that results were persisted in stable storage (e.g. at a transactional message queue) before committing the DSMS transaction. This would provide transactional behavior for the entire path between the stream source and the client. This architecture is described further in Section 5.

3.5 Comparison with Database Systems

For the purposes of this paper, the most significant differences between data streams and databases are:

- Streams are append-only, and queries on streams are read-only. Therefore, there is no need to isolate DSMS clients from the actions of concurrent users, unless the client also accesses relations.
- Stream are inherently ordered, unlike relations.
- While a DBMS executes transactions that consist of sequence of snapshot queries, a DSMS executes a collection of relatively independent analysis queries. These queries are continuous, long-lived, and typically executed via shared processing.
- A DSMS is *data-oriented*, whereas a DBMS is *operation-oriented*: rather than applying queries to a static collection of data, a DSMS "applies" incoming stream tuples to update the result sets of the current set of continuous queries.
- Most of the interesting state in a DSMS is resident in memory, not on disk. Therefore, durability is more about checkpointing the current runtime state of the system than it is about ensuring that the on-disk state of the system is consistent.

3.6 Prior Work

There is an extensive literature on data streams, which we do not attempt to summarize here. [GÖ03] is a recent survey of the field. In particular, crash recovery and high availability have been previously studied for data stream systems (e.g. [SHCF03, HBR+05]), but not in a transactional setting.

Active databases are among the intellectual predecessors of stream query processors. The HiPAC project proposed a framework for active databases that included a "generalized transaction model for defining correctness of concurrent execution of user transactions and triggers".[DBB+88] HiPAC allows the execution of user transactions, triggered actions, and "checkers" that validate trigger conditions, so a transaction model is necessary to define the ways in which these operations can be legally interleaved. [DHL90] discusses the integration of triggers with extended transaction models, such as those discussed in Section 2.2. Both HiPAC and the work of Dayal et al. differs from the current paper in that prior work on active databases is still operation-oriented, rather than data-oriented: transactions are used to provide guarantees about the concurrent evaluation of triggers and their triggering actions.

4 Case Studies

Now that we have introduced both transactions and data streams, we turn to several case studies of how data stream management systems could stand to benefit from transaction-like functionality. We consider examples that focus on isolation, durability, and crash recovery, in Sections 4.1, 4.2, and 4.3, respectively.

4.1 Isolation

A continuous query sees a *snapshot* of the state of the database when it begins execution. In this section, we consider how a continuous query's snapshot should change over time. This requires defining an isolation model for continuous queries.

As a concrete example of the need for such a model, we consider the problem of isolation for mixed joins. In a system that supports both snapshot and continuous queries, it is possible for other database clients to modify a relation that participates in a mixed join. An isolation model is required to define the way in which these modifications are reflected in the output of the mixed join. In this section, we first discuss traditional approaches, and then propose *window isolation*, an isolation model suitable for continuous queries.

4.1.1 Traditional Isolation Models

Traditional approaches to transaction isolation do not offer a satisfactory solution. A naive approach would be to implement each continuous query as a single transaction. The conventional transaction isolation rules would require that a continuous query be isolated from any changes to the database that have been made since the query began execution. Therefore, any modifications to the relation in a mixed join never be made visible to continuous queries that began execution before the modification took place.

Using a single transaction is clearly suboptimal for the kinds of long-running, continuous queries that are typical in a data stream system. Most client applications would expect that modifications to base relations will *eventually* be reflected in the output streams of continuous queries. In the example query given in Algorithm 2, the set of "interesting" stocks might change periodically, and at some subsequent point the output of the mixed join should be adjusted accordingly.

Of course, clients can stop and restart their continuous queries in order to receive refreshed results, but this is also unsatisfactory. Because clients in a data stream system are often only loosely coupled to queries (Section 3.4), so it may even be impossible for clients to restart queries on demand. The client might also miss any query results that would have been emitted during the time that the query is being restarted, which is unacceptable for some applications. This approach would also be difficult to implement efficiently: in order to ensure that a continuous query sees an immutable snapshot of each base relation, the query would typically need to hold a lock or similar construct on every table it accesses for the duration of its execution.

Rather than evaluating a continuous query in a single database transaction, another naive approach would join each new stream tuple against the "most recent" version of the table. This solves the visibility problem: modifications to a table will promptly be made visible to any mixed joins that involve the table. However, this alternative is also problematic. The result of a mixed join could be derived from multiple snapshots of the join relation, so join results may become inconsistent. In the example query given in Algorithm 2, if a row is removed from the table of "interesting" stocks in the midst of a window, an incomplete sum will be included in the next window's worth of results from the mixed join. Similar problems occur if a row is added to the relation in the midst of a window. In both cases, the problem is exacerbated by the fact that the user has no way to observe that data has been silently omitted from the result of the join.

Another problem with joining each stream tuple against the most recent version of the join relation is that it effectively limits the choice of join algorithm to nested loops. Using a join algorithm such as a hash join that computes a temporary data structure from the join relation would require synchronizing the temporary structure with the relation from which it was derived. While this is possible, it would represent an additional headache for implementers and likely hurt performance.

4.1.2 Window Isolation

We propose using the mixed join's window as its "unit of visibility". That is, during the computation of a single window's worth of results, the mixed join accesses an immutable, consistent snapshot of all the relations in the system (presumably, the same snapshot that a new transaction would see if it began at the same instant that the window began). At each new window boundary, a new snapshot of the system is taken, which incorporates any modifications into the next window's worth of results from the mixed join. We call this model *window isolation.*

This provides an adequate solution for the mixed join example given in Algorithm 2. When the set of "interesting" stocks is modified by a transaction that successfully commits, the modification is ignored for the duration of the current window. The next window includes *only* the new set of interesting stocks. Assuming that each transaction modifying the `stocks_of_interest` table ensures that it is left in a consistent state, the mixed join's results in any given window will be self-consistent.

Note that this approach works for both overlapping and disjoint windows. In an overlapping window, a stream tuple can be part of two or more windows in the same query. In this case, the two "versions" of the stream tuple will need to be joined against two different snapshots of the join relation.

Algorithm 3 An example of a derived stream and a continuous query in CQL. They use differing window clauses, but access the same relation ("users").

```
CREATE STREAM user_purchases AS
    SELECT ISTREAM(u.name as user_name,
                   s.name as stock_name,
                   t.price, t.volume)
    FROM stocks_of_interest s,
         users u,
         ticker t
       [ RANGE ‘5 minutes’ SLIDE BY ‘5 minutes’ ]
    WHERE s.symbol = t.symbol
      AND s.user = u.name
      AND t.purchaser = u.name;

SELECT RSTREAM(up.user_name,
               sum(up.price * up.volume) + u.bank_account)
FROM users u,
     user_purchases up
   [ RANGE ‘1 hour’ SLIDE BY ‘30 seconds’ ]
WHERE u.name = up.user_name
GROUP BY up.user_name;
```

4.1.3 Composition of Windows

This model is applicable to more complex queries. Suppose that the output of a mixed join is used in another continuous query. For example, a relation-to-stream operator might be applied to the mixed join to yield a derived stream. That derived stream might itself be used as the stream operand to another mixed join, with a window clause that differs from the window clause of the original mixed join. In this situation, how should multiple mixed joins in this query be isolated from changes to their base relations?

A critical assumption is that derived streams are treated as *independent* by the DSMS. That is, the DSMS does not make a fundamental distinction between the tuples emitted by a derived stream and the tuples contained in a base stream: both are merely sequences of independent tuples. The tuples in a data stream have no fundamental grouping: they are organized into windows by the window clause of a given query that accesses the stream. This assumption justifies our policy that the snapshots seen by these two mixed joins depend on *only* on their own window clauses.

Algorithm 3 contains an example query that illustrates a potential problem with this model. The `user_purchases` derived stream contains stock purchases made by a group of users (stored in the `users` table). Each user has an associated set of "interesting" stocks, which as before is stored in the `stocks_of_interest` table. The `user_purchases` derived stream can be used in other queries. For example, Algorithm 3 also includes a continuous query computes the "total wealth" of each user, by summing their recent stock purchases with their bank account balance.

This example is problematic, because the `users` table is joined twice: because their window boundaries differ, the continuous query and the derived stream will see different snapshots of the `users` table. This may yield inconsistent or unexpected results.

It is also possible for a single query to contain multiple, differing window clauses. For example, a query might contain two independent mixed joins, the outputs of which are then joined. As with derived streams, we treat the two mixed joins as independent: the database snapshot seen by each mixed join depends on its window clause.

4.1.4 Implementation Considerations

Window isolation requires that a new snapshot of the state of the database be taken at every window boundary. For example, if another client attempts to modify a relation that participates in a mixed join, the DSMS must ensure that the mixed join's snapshot remains valid for the duration of its current window. Broadly speaking, there are three ways to do this:

1. Prevent the modifying transaction from committing until the end of the mixed join's current window

2. Allow the modifying transaction to commit, but only join the window against the relation at the end of the current window boundary

3. Allow the modifying transaction to commit, but ensure that the mixed join's snapshot of the join relation remains unchanged

The first alternative is clearly inefficient, since windows that last for many minutes or hours are not uncommon. This approach is even more infeasible if a single relation participates in several mixed joins with differing windows. The second alternative is also undesirable, because it disallows the incremental computation of the mixed join's result set. As described in Section 3.2.2, in many circumstances the DSMS cannot afford to buffer the entire window's worth of data in memory.

Therefore, practical implementations must ensure that a mixed join's snapshot remains unchanged in the face of committed modifications to the join relation. There are two ways in which this might be achieved:

1. The mixed join could materialize the join relation into a temporary table at the beginning of each window boundary

2. The DSMS could use a concurrency control scheme in which readers do not block writers; this allows the mixed join to continue to read from its snapshot of the join relation without preventing modifying transactions from committing

The first alternative is impractical if the join relation is large, or window boundaries occur with any degree of frequency. Therefore, window isolation is best implemented in systems that use a concurrency control scheme that allows readers to not block writers. Unfortunately, most traditional lock-based concurrency control techniques do not satisfy this property. Multi-version concurrency control (MVCC) is the most popular concurrency control scheme that provides this capability[BG83].

Note that in this scheme, a single continuous query uses multiple snapshots over the duration of its execution. If overlapping windows are used, a single continuous query may even use multiple, inconsistent snapshots of a table at the same time. This is in contrast to traditional isolation models, which usually require that a single query use a single, consistent snapshot at any given time. While this would be challenging to integrate into a standard lock-based concurrency control scheme, it should be possible in theory to extend a typical implementation of MVCC to allow the query processor to view different snapshots of the table when evaluating different nodes in the graph of operators.

In a typical implementation, the continuous query itself executes inside a "super-transaction", and "micro-transactions" are periodically initiated in order to obtain a new, consistent snapshot of the join relation. It is worth noting that these micro-transactions are *not* nested transactions in the usual sense, since a nested transaction inherits the snapshot of its parent transaction. This model is closer to the "sagas" technique described in [GMS87]: in the terminology of that paper, the super-transaction in which the continuous query executes is analogous to a saga, and the sequence of micro-transactions are effectively normal transactions.

4.2 Durability

Archived streams are another approach to combining current and historical data in a DSMS[CF04]. An archived stream is a stream whose content is written to disk and can be subsequently accessed by queries. As new stream tuples arrive, the DSMS updates the result sets of any relevant continuous query, and then periodically adds the stream tuples to the archived stream for later access. Because archived streams are stored on disk, they should be persistent in the event of a system crash. This raises durability concerns: how often should stream tuples be persisted to an archived stream, and what guarantees should the DSMS make about the durability of archived streams?

Again, we use the window concept as the unit of consistency: as each window of the archived stream is produced, the database ensures that all the tuples in that window have been flushed to disk (either by force-writing the tuples themselves, or flushing a log record describing the necessary modifications). This ensures that windows are archived in an atomic fashion: either all the tuples in a window will be persisted in the event of a system crash (if the window boundary has already passed), or none of them will.

If window boundaries occur infrequently, the DSMS may not be unable to keep all the tuples in a window in memory until the end of the window. Furthermore, keeping stream tuples strictly in memory until the next window boundary produces highly "bursty" I/O usage: the DSMS will likely overload the I/O subsystem at window boundaries, but leave it largely idle otherwise. Therefore, a practical implementation of this model would write tuples to the archived stream as they arrive, or spool them to the archived stream in the background. At the next window boundary, the DSMS does the equivalent of a transaction log flush: all remaining stream tuples are flushed to disk, and then a transaction log record is written to record the end of the window. During crash recovery, the recovery component of the DSMS should remove all the tuples from the suffix of an archived stream that do not have an associated "commit" record in the transaction log.

Note that in database systems, the tuples modified during a transaction are often randomly distributed over the disk. Due to the relatively high cost of random I/O, it makes sense to commit transactions by force-writing *only* the transaction log to disk, since this requires only sequential I/O, and to allow the random I/O performed by the transaction to be incrementally written to disk over time. This is not the case with archived streams: all I/O on the archived stream consists of sequential append operations, so there is no benefit to writing a separate transaction log describing each incremental change.

A related problem isolation problem arises when we consider the case of an archived stream that is accessed (as a relation) by a snapshot query. We suggest that the snapshot query should see a version of the archived stream that corresponds to the most recent window boundary, at the time the snapshot query began execution.

4.3 Crash Recovery

Most of the interesting state in a database system resides on disk. While runtime state is kept in memory for performance reasons, the system arranges for all modifications made by committed transactions to be persisted to disk. In most database applications, any queries that were in progress at the time of a system crash can safely be aborted. Therefore, crash recovery in database systems is largely a question of restoring the on-disk state of the database to consistency: once that is done, in-memory state can either be reconstructed from the disk contents, or else freshly initialized. Because snapshot queries are inherently transient, any in-progress snapshot queries can reasonably be aborted during crash recovery.

In contrast, most of the interesting state in a data stream system is resident in memory. In many cases, data is never written to disk, because it arrives too quickly and is of relatively short-term interest[GÖ03]. Furthermore, a long-running continuous query may contain a significant amount of state: for example, an aggregate function computed over a data stream can depend on all the previously-seen tuples in the stream. Merely aborting and restarting the query anew if the system fails is undesirable, because it essentially destroys all this state. Clearly, most data stream applications would prefer that continuous queries be persistent over system crashes. Many DSMS applications cannot afford to miss any data, even if the DSMS crashes. This raises the question: how do we perform correct crash recovery in data stream systems?

4.3.1 Window-Oriented Crash Recovery

If we model a DSMS as a continuous computation that transforms inputs (data streams) to outputs (result streams) by applying a set of operators (continuous queries), implementing correct crash recovery requires checkpointing the state of this computation at various points. To reproduce the state of a computation at time t, we require two pieces of information:

1. The state of the computation at some time t_0 in the past
2. All the data that entered the system between t_0 and t

We can use a process analogous to replaying a transaction log to restore the computation's in-memory state.[2] We now briefly discuss how these two pieces of information can be recorded in a practical DSMS.

The state of the computation can be recorded by serializing the stream processor's graph of operators to disk. Each operator has some associated in-memory state that captures part of the state of the computation. For example, an aggregation operator must maintain the running value of the aggregate. The DSMS can periodically checkpoint this in-memory state by flushing a copy of it to disk. The system should take a new snapshot when it reaches a new point of consistency: roughly speaking, at window boundaries. As with transaction logs in a DBMS, we assume that there is a stable location (e.g. disk) to store the checkpoint data in the event of a crash.

To ensure satisfactory performance, two techniques are helpful. First, the DSMS need only record the incremental difference between the current state of the operator and the state at the last checkpoint. This significantly reduces the amount of I/O required. Second, the DSMS can take steps to ensure that the checkpointing operation can be interleaved with normal execution. This is important, as many practical data stream management systems cannot afford to block stream processing while waiting for disk I/O to complete. This interleaving could be implemented by simply making a copy of the in-memory state of the operator graph, and then using the copy to form the checkpoint. However, a more fine-grained approach is probably necessary. In theory, by examining the set of continuous queries, their associated window clauses, and the operators used to implement the queries, a graph of operators could be split into sub-graphs that can be checkpointed independently.[3] This would allow checkpointing to be done in parallel, and would also allow the other components

[2]This assumes that the computation is *deterministic*: the future state of the computation is a total function of the computation's previous state and any additional input data. This does not hold for probabilistic computations or queries with side effects, but we do not consider such queries here.

[3]As noted in Section 3.3.1, shared processing can complicate correct behavior: because a single operator can be used to implement multiple queries, dividing the operator graph into independent sub-graphs can be complex.

of the operator graph to continue query processing while a sub-graph is being checkpointed.

4.3.2 Buffered Streams

In addition to checkpointing the state of the system, a DSMS must also have access to any data that entered the system between the time of the last checkpoint and the time of the crash. In addition, most systems will want to also retain any data that has arrived since the system crash.

The DSMS itself cannot maintain this data: data typically arrives too rapidly to be written to disk on-the-fly[CCD$^+$03]. Instead, we assume that there is a stable *buffer* associated with each stream input source. The buffer holds the most recent k stream tuples, and is assumed to be resilient to system failures. In practice, this could be implemented by a simple replication scheme, in which n copies of each incoming stream tuple are made, each stored in a different buffer. Traditional techniques for replication and high-availability can be used to make such a system quite robust, although perhaps with non-negligible overhead.

In a traditional DSMS, the data stream input protocol is *unidirectional*: a data source connects to the DSMS and sends data to the system's stream input manager. When using buffered streams, the DSMS uses a bidirectional communication protocol, in which the DSMS acknowledges receipt of a stream tuple when a point of consistency has been reached. This allows the buffered stream to discard the data the DSMS has acknowledged.

The DSMS sends acknowledgments to the stream buffer when it is certain that a given stream tuple can safely be discarded. For example, an acknowledgment could be sent for a tuple after routing the tuple to all the nodes of the operator graph, and then checkpointing the state of those operators. Clearly, there is a tradeoff between the work required to perform frequent checkpoints, and the memory required for large stream buffers. Many applications would be willing to accept weaker crash recovery guarantees in exchange for reduced checkpoint requirements, so we suggest that this tradeoff be exposed to the user.

It is worth noting that the DSMS only requires stable storage for a finite amount of data to ensure correct recovery: the state of the operator graph, and a bounded number of recent stream tuples. The relatively small size of this data suggests that it could also be stored in a stable medium with high-performance but small storage capacity, such as flash memory or a solid-state disk.

4.3.3 Summary

In this subsection, we have sketched the design of a window-oriented crash recovery mechanism for a DSMS. A key problem is determining points of consistency, to record the state of the system to disk, and to allow stream tuples to be acknowledged. Defining this crash recovery technique more completely and determining whether it is practical is an area for future work.

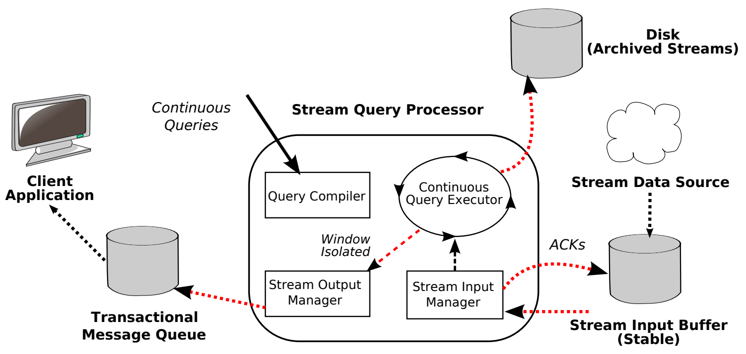

Figure 2: A hypothetical architecture for a transaction-oriented DSMS.

5 The Role of Transactions in a DSMS

In database systems, transactions are *operation-oriented*: they are used to organize sequences of related operations into an atomic group. Much of the criteria for the correct execution of a transaction can be captured in a formal isolation model, which specifies the transaction schedules that are consistent with the system's isolation guarantees.

A DSMS does not execute "sequences of related operations", so it follows that transactions will not play the same role in data stream systems that they do in database systems. As noted in Section 3, data stream processors are data-driven, not query-driven: query execution in a DSMS often takes the form of pushing stream tuples through a graph of operators, then discarding the tuple. The DSMS is largely free to choose the order in which tuples visit nodes of the operator graph.

Therefore, we suggest that transactions are useful for providing guarantees about the *movement of data* into, within, and out of a DSMS. As described in Section 4.1, the isolation of continuous queries is based on window boundaries, which essentially defines "points of consistency" at which new snapshots of the database can be taken by the continuous query. This effectively specifies constraints on the movement of data within the DSMS. Sections 4.2 and 4.3 discuss the application of transaction-like concepts to the movement of data into the system (from the stream source), to the disk (archived streams), and then eventually the delivery of query results to the client or an intermediate middleware system.

Figure 2 describes a potential architecture for a transaction-oriented DSMS. Transactional guarantees are provided for all the data movement within the system: input from (buffered) data streams, to data movement within the DSMS itself, to the eventual output of result streams to a transactional message queue. This architecture simplifies the construction of reliable and correct streaming applications: application programmers can reason more easily about how the system isolates one query from another, and how it will recover from crashes.

To an external observer, such a system takes "atomic" steps within the data stream: it accepts a window's worth of input, performs internal computations, and then pushes a window's worth of results to the system's output interface. The "transaction" covers the entire lifecycle of a group of related tuples, from their arrival in the system to their delivery to the output queue.

6 Conclusions and Future Work

In this paper, we have investigated several ways in which the transaction concept can be applied to data stream management systems. We began by presenting background material in Sections 2 and 3, focusing on transaction models for long-running tasks and the differences between database systems and data stream systems. In Section 3.2.2, we also introduced the *mixed join* as a convenient way to express queries that combine live stream data with static relational data, a requirement that arises frequently in practice.

In Section 4, we proceeded to discuss several situations in which isolation, durability, and crash recovery in data stream systems would be simplified by the addition of transaction-like behavior. The concept of dividing a data stream into windows and maintaining consistency within a window shares some similarity with techniques like sagas[GMS87], which divide long-lived transactions into a collection of related "micro-transactions". As discussed in Section 5, the essential difference is that data stream systems are data-oriented rather than operation-oriented: windows serve to define sub-sequences of related tuples within a data stream. Section 5 also sketches a model for a transaction-oriented DSMS, in which all data movement within the system is subject to transaction-like guarantees. The proposals made in this paper are exploratory by nature: an area for future work is evaluating which of these ideas are practical, and then defining them with the appropriate degree of rigour.

We began this work with the intention of defining a transaction model for data stream processors that was conceptually similar to the formal transaction isolation models discussed in Section 2.3. While some kind of formal model for DSMS transactions would still be desirable of course, it is now clear that the traditional isolation model concepts of serializability and constraints on transaction schedules are inappropriate for data stream systems. Defining a more appropriate and complete transaction model for data stream systems remains an area for future work. An open question is whether any single formalism can play the same central role that isolation models play in database systems. The formulation of a concept analogous to transaction serializability in a database system would be a major step toward resolving this question.

References

[ABW03] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[ALO00] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 67–78, March 2000.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[BDG+94] A. Biliris, S. Dar, N. H. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 44–54, 1994.

[BG83] P. Bernstein and N. Goodman. Multiversion concurrency control—Theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[BGS01] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, 2001.

[BSW88] C. Beeri, H. J. Schek, and G. Weikum. Multi-level transaction management: theoretical art or practical need? In *Lecture Notes in Computer Science*, volume 303, pages 135–154. Springer-Verlag, 1988.

[CCC+02] D. Carney, U. Cetinternel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *Proceedings of the International Conference on Very Large Databases*, pages 215–226, 2002.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, pages 269–280, 2003.

[CF04] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proceedings of the International Conference on Very Large Databases*, pages 348–359, 2004.

[CJSS03] C. Cranoe, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *Proceedings* *of the ACM SIGMOD International Conference on Management of Data*, pages 647–651, 2003.

[CR94] P. Chrysanthis and K. Ramamrithan. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[DBB$^+$88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhuri. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, 1990.

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Compututing Surveys*, 35(2):114–131, 2003.

[GLPT76] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modeling in Data Base Management Systems*, pages 365–395. North-Holland, Amsterdam, The Netherlands, 1976.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, May 1987.

[GÖ03] L. Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2), 2003.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, September 1981.

[Gra95] J. Gray. Queues are databases. In *Proceedings of the 7th High Performance Transaction Processing Workshop*, 1995.

[HBR$^+$05] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of The International Conference on Data Engineering*, April 2005.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[HMPJH05] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[KNV03] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 341–352, 2003.

[KWF06] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 623–634, 2006.

[LS03] A. Lerner and D. Shasha. AQuery: Query language for ordered data, optimization techniques, and experiments. In *Proceedings of the International Conference on Very Large Data Bases*, pages 345–356, 2003.

[Mos85] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[PGB$^+$04] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-Keller, and E. W. Biersack. Using data stream management systems for traffic analysis—a case study. In *Proceedings of Passive and Active Measurements*, April 2004.

[PK88] C. Pu and G. E. Kaiser. Split-transactions for open-ended activities. In *Proceedings of the International Conference on Very Large Databases*, 1988.

[SHCF03] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of The International Conference on Data Engineering*, pages 25–36, 2003.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[SW04] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 263–274, 2004.