Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 742

Systems Group, Department of Computer Science, ETH Zurich

Building Data Flows Using Distributed Key-Value Stores

by
Martin Hentschel, Maxim Grinev, and Donald Kossmann

October 2011

inf | Informatik
Computer Science

Building Data Flows Using Distributed Key-Value Stores

Martin Hentschel Maxim Grinev Donald Kossmann
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
{hemartin, grinevm, donaldk}@inf.ethz.ch

ABSTRACT

Social communication features on most of today's largest websites require propagating the data inside the database/ key-value store leading to massive data flows. In this paper we study alternative architectures to build data flows using distributed key-value stores. We compare programming model, execution model, failure model, and scalability highlighting a problem of the state-of-the-art architecture based on an external queue: non-optimal resource utilization. As part of this study, we propose an optimization of this approach by integrating queues into the key-value store. It results in better resource utilization and, thus, more cost-effective scalability; as well as easier programmability and lower maintenance cost. Our experimental study confirms these findings.

1. INTRODUCTION

Social communication features on most of today's largest websites require propagating the data inside the database. Incoming data is stored redundantly for efficient retrieval. Such data propagation leads to massive data flows. For example, on Twitter, users follow each other's status updates called tweets. Each new tweet posted by a user initiates a complex data flow that propagates the tweet to the user's followers, inserts it into Twitter lists, and indexes it for full text search [1]. All tweets of Twitter's millions of active users have to be pushed through the system creating massive data flows.

Typical properties of such data flows are that they are append-only, as input and propagated data items will not overwrite existing data; each data item can be uniquely identified by a key; and observing temporal inconsistencies is not a problem. Temporal inconsistencies occur when input data is already stored in the database but has not been completely propagated. For example, if the user submits a tweet they should immediately see it on their page but it may take some time to appear on the pages of their followers.

Building a data flow system is not trivial. There are many requirements to the system that need to be considered. (1) The system has to sustain a high throughput of incoming data. (2) Each user request should be answered within a low and guaranteed response time even in presence of skewed workloads. That is, the size of the data flow should not have an impact on response time. For example on Twitter, the number of followers of users varies a lot. Celebrities typically have many more followers than common people. It leads to different amounts of data that need to be propagated depending on the user. But still, we need to guarantee low response time for every user. Response time is defined as the time until a user request is acknowledged by the system. (3) The system should be robust enough to handle short periods of bursts of incoming data. For example, during Barack Obama's inauguration Twitter experienced five times the normal workload for several minutes [14]. (4) Fault tolerance is an important requirement as we cannot afford loosing data. For example, if a tweet is sent it should (eventually) be delivered to all the followers. (5) The system should be scalable in terms of throughput to handle a growing user base.

On the other hand, the mentioned properties of data flows simplify the system implementation. (1) It is not required that the system implements exactly-once semantics. Because every item can be uniquely identified by its key, repeated insertion will always overwrite already existing data. It means that every write is idempotent. Therefore it suffices that the system implements at-least-once semantics, which is less expensive than exactly-once [4]. (2) Concurrency control between threads in the system that propagate the data is not needed. Because the data flows are append-only, and there are no in-place updates, synchronization is not required. This is obviously not correct for data flows that exceed data propagation characteristics. For example, analytical data flows require incremental updates of aggregate values and therefore concurrency control mechanisms. (3) Because temporal data inconsistency is not an issue, the system may propagate data asynchronously allowing lower and guaranteed response times to acknowledge user requests. (4) Last, the properties of data flows allow to do without ACID transactions in the database system: atomicity is achieved using repeated idempotent writes, consistency is relaxed, and concurrency control is not needed. It allows us to built data flows on top of modern key-value stores, like Cassandra [8], that typically do not provide ACID transactions but are popular to build Web applications. (In this

paper we will use the terms *database* and *key-value store* interchangeably.)

There are several approaches to build data flows. The first and naïve approach executes all steps of the data flow synchronously. As we will show in this paper, this approach sustains high throughput and provides good scalability but fails to guarantee response time and cannot handle bursts. The second approach is based on an external queue and workers to buffer and execute incoming requests asynchronously. Among others, it is used in Twitter [10]. As we will show in this paper, it overcomes the drawbacks of the naïve approach in that it guarantees low response time and handles bursts. However, it needs increasingly more nodes (for application and queue) in order to scale. We propose a third approach that tightly integrates the queue into the key-value store: the queue is partitioned across the nodes of the key-value store. It results in better resource utilization compared to the external queue approach because it reduces network traffic and better utilizes the CPUs of the nodes. Traditionally, products combine database and queue components for consistency reasons and easier application development [16, 9], but do not tightly integrate these components for better resource utilization.

In this paper we study the above listed approaches to build data flows using distributed key-value stores. We focus on the programming model, execution model, failure model, and scalability. We compare performance characteristics of all approaches varying essential workload parameters. We conclude that the third, integrated approach provides a number of advantages, the most significant being better resource utilization and therefore reduced cost of scalability in comparison to the external queue approach. Furthermore, the integrated approach offers an easier programming model and has lower maintenance cost.

Although this work is inspired and illustrated by examples of social networks, the techniques considered in this paper can be used for any data flows that have the above listed requirements and properties.

The remainder of this paper is organized as follows. Section 2 presents an example data flow we will use throughout the paper. Sections 3, 4, and 5 discuss the synchronous, external queue, and integrated approaches respectively. Section 6 compares these approaches experimentally. Section 7 discusses related work. Section 8 concludes this paper.

2. RUNNING EXAMPLE

To discuss the alternative approaches to build data flows, we use a simple data flow as running example. We will discuss a more sophisticated data flow in our experimental study. In this simple data flow, illustrated in Figure 1, a user posts a data item I_A to an application. The application requires I_A to be written before the user request is allowed to return. We say that I_A is *session-critical* as the write of I_A should be session-consistent: within a session the user should see their own writes. Furthermore, data item I_A causes two more items I_B and I_C to be inserted into the database. I_B and I_C are not session-critical and may be written asynchronously in the background after the user request has returned. As discussed in the introduction we will assume that I_A, I_B, and

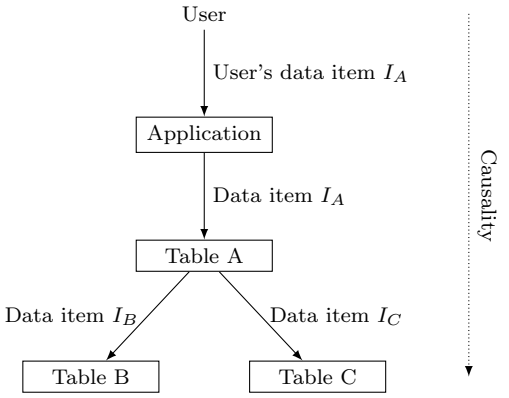


Figure 1: Example Data Flow.

I_C can be uniquely identified by key. It allows implementing the insertions as idempotent operations.

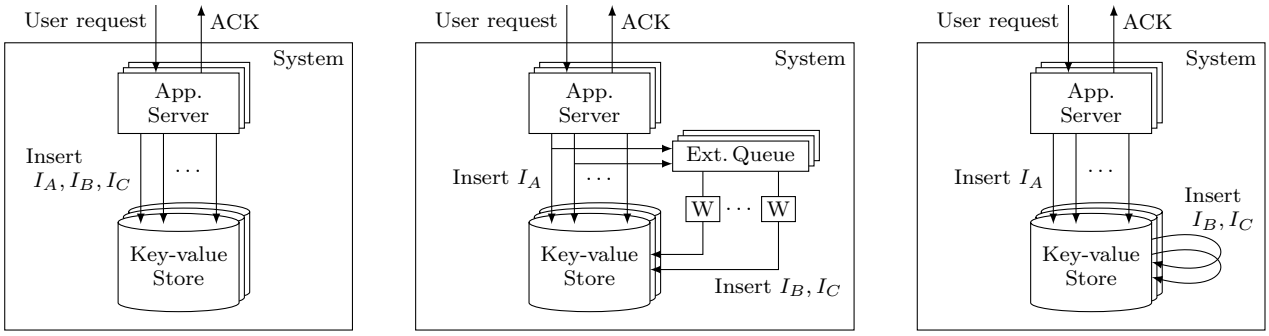
In Twitter, for example, data item I_A might be a tweet posted by a user. The application inserts the tweet into the user's timeline. (In Twitter, the timeline is a list of tweets by the user and the users he follows.) I_A is session-critical because the user should see that the tweet was inserted correctly. The tweet is then forwarded to the (two) followers of the user by inserting items I_B and I_C into the respective timelines of the followers. I_B and I_C are not session-critical because the user does not need (and want) to wait for the successful delivery of the tweet to his followers.

The causality aspect between session-critical writes (I_A) and non-critical writes (I_B and I_C) is important; as is the fact that they will typically have different complexities. Item I_A causally determines items I_B and I_C. Therefore, it is sufficient to remember I_A; I_B and I_C can be recovered from I_A at any time. Session-critical writes and non-critical writes typically have different complexities. For example, inserting a tweet into the system is cheap, but forwarding it to hundreds of thousands of followers may be much more expensive. This rules out straightforward ideas such as writing all items in a single batch (i.e., the naïve approach discussed below).

3. SYNCHRONOUS APPROACH

The first and naïve approach to implement this data flow is to execute all steps of the data flow synchronously, in a single batch. This is the standard setup of a web application that uses, for example, an Apache web server and a MySQL database. The application will write all data items I_A, I_B, and I_C to the database and only after the last item has been written the initial user request will return. Figure 2(a) displays the synchronous approach to execute the example data flow from Section 2. The application uses multiple threads to write to the database. A thread writes I_A, I_B, and I_C to the database. After the write has finished the initial user request will be acknowledged.

The advantage of the synchronous approach is that it is easy to use, no modifications or extensions to the application or database are needed. The disadvantage of this approach is the high response time of the user request. Despite the



(a) Synchronous Approach (b) External Queue Approach (c) Integrated Approach

Figure 2: Alternative Architectures to Build Distributed Data Flows.

rather small session-critical write (I_A) the user will have to wait for a long time until all other non-critical writes (I_B and I_C) have been completed. This is particularly harmful for social web applications in which a user request typically causes many successive or long-running writes to the database to propagate the data to all the user's connections in the social network.

3.1 Programming Model

The synchronous approach requires no special programming model. All updates can be executed sequentially from a standard database client application. It is simple to use from from an application developer's perspective, but it lacks the expressiveness to, for example, describe that only I_A is a critical update.

3.2 Execution Model

The execution model is based on the standard web application architecture with no additional components and no changes to the database.

3.3 Failure Model

In case of failures the developer needs to take care that there are no inconsistencies. Inconsistencies occur, for example, when a failure happens after items I_A and I_B have been written successfully but before I_C has been written. The failure will be visible to the application because the request to insert I_C will return an error, or not return at all and the application will experience a time out. In this case there are two options to fix it: (1) write missing items (i.e. I_C in this example) again and rely on idempotent writes or (2) delete all items that have been written so far. Both cases are easy to implement because we assume that I_A, I_B, and I_C can be uniquely identified by key.

3.4 Scalability

This approach is easy and cost-effective to scale using standard techniques supported by modern application servers and databases, for example, load balancing and data partitioning.

4. EXTERNAL QUEUE APPROACH

The high response time of the synchronous approach can be reduced by only acknowledging the session-critical write to the user. All non-session-critical writes are executed asynchronously after the acknowledgement. The state-of-the-art approach to implement asynchronous execution is to use an external queue. The user request is buffered in the queue (in addition to executing the session-critical write) before sending the acknowledgement to the user. The queues are drained by worker threads. The worker threads insert the appropriate non-session-critical writes into the database. The user request can return much faster compared to the synchronous approach. The disadvantage is temporal data inconsistency, but as we discussed in the introduction that is not an issue for propagational data flows. Other disadvantages of this approach are (1) higher cost for setup and maintenance as it requires an external queue and workers, which typically reside on separate machines; and (2) more complicated programming model and failure handling as we discuss below.

Figure 2(b) displays the differences to the previous approach. An external queue is added to the system. The application layer writes I_A to the database and, potentially in parallel, pushes I_A to the queue. After these two steps the application is able to acknowledge the user request. Worker threads (denoted by $\boxed{\text{W}}$ in Figure 2(b)) read I_A from the queue and write I_B and I_C to the database.

4.1 Programming Model

The external queue allows the application developer to express which items are session-critical (by writing them to the database immediately) and which items are not session-critical (by storing them in the queue). This expressiveness comes at a significant price: queue handling must be programmed out explicitly by the developer, including worker threads and failure handling.

The application developer must obey the following rules when using an external queue. First, the application has to store data item I_A in the queue after successfully writing it to the database. Second, the application programmer has to implement worker threads that read I_A items from the queue and write the corresponding items I_B and I_C to the database. To optimize response time, I_A can be stored in the queue in parallel to the write of I_A to the database. (We used this optimization in our experiments in Section 6.)

4.2 Execution Model

There need to be additional machines for queue and workers and make them interact as described above. For small workloads, worker threads can be run on the same machine as the application. But for real-world applications and for larger workloads, they have to run on separate machines in order to avoid overloading the application server as it will be shown in our experiments. As in the synchronous approach the execution model does not require any changes to the database.

4.3 Failure Model

The developer has to fight failures at two fronts, at the application layer and within worker threads. The application writes the session-critical data item I_A to the database and to the queue in parallel. That means, if the write to the database results in an error or we do not get a response, we may have already pushed I_A to the queue. Vice versa, if the write to the queue was unsuccessful, we may have already written I_A to the database. Therefore we can only retry writing to the database or to the queue in case of failures. Otherwise it would lead to an inconsistent state of the data: I_A has been written but the corresponding I_B and I_C will never exist—or I_A has *not* been written but I_B and I_C exist. Thus, we can only report an error to the user in the case when both writes, to the database and to the queue, fail.

Once the critical data item has been pushed to the queue it must be sure that the non-critical data items I_B and I_C will be written to the database under any circumstances. Within worker threads, two different failure cases need to be considered. First, the worker thread quits working after it has read the item from the queue and thus was not able to complete the job. Another thread cannot repeat the work of the failed thread, because item I_A has been deleted from the queue. Items I_B and I_C are lost. Second, the worker thread is not able to write to the database because of either an error response or no response at all.

To deal with the first failure of lost items, queue implementations typically allow to read items *tentatively*. A worker thread reads an item tentatively and if the read is not confirmed by the worker at a later point in time, the item will stay in the queue and be visible again after a specific timeout interval. Thus, another worker is able to repeat the job. In between the tentative read and the confirmation the item cannot be read by any other workers to prevent multiple workers from doing the same work. If a worker crashes after it has written some or all non-critical data items to the database but before the confirmation, another worker worker will read the non-confirmed data item from the queue and repeat the writes of the non-critical items. As in all approaches in this paper, we rely on idempotency to deal with duplicate writes.

To deal with the second failure in which the worker thread does not crash but the writes to the database respond with an error or with no response at all, the worker thread needs to retry the writes until successful. The worker thread cannot inform the application of such an error because the originating user request has already been acknowledged. Therefore it remains to the worker thread to retry writing to the data storage until successful (or log the error).

4.4 Scalability

To support asynchronous execution we need additional queue and worker components. Therefore, scaling this approach means to increase the number of nodes not only for the application and database but also for the queue and workers. As result it requires increasingly more resources, in particular network and CPU, to scale this approach. Fortunately, this drawback can be eliminated by the optimization described in the next section.

5. INTEGRATED APPROACH

In this section we describe an optimization of the external queue approach that aims at better resource utilization and provides advantages in development and maintenance of the system. There are two basic design decisions behind this approach.

First, we distribute and embed the queue across the nodes of the key-value store (i.e., database). Each database node maintains its own queues and worker threads that execute data flow tasks. When an item that should be propagated is written to the database, the database node executing the write submits the corresponding task into its local queue. The task is later executed by one of the node's local workers. Distributing queues and workers across database nodes removes the network cost of communicating with the external queue and shifts work to the under-utilized CPUs of the database nodes. Because the execution of the data flow is not CPU intensive, we can leverage free resource of database nodes and utilize resources more efficiently. We prove this point in the experimental section of this paper. Furthermore, integrating queues into the key-value store eliminates maintenance and administration cost for additional products such as an external queue.

Second, fault tolerance for guaranteed asynchronous execution is implemented by reusing the replication mechanism of the distributed database. Replica nodes remember the task and execute it in case when it was not successfully executed by the original database node. Piggybacking fault tolerance on database replication allows to reduce the communication overhead within the database.

Figure 2(c) illustrates the integrated approach. The execution model of this approach, within the distributed database, is shown in Figure 3. For simplicity, replication and failure handling is not shown in Figure 3. It highlights that, in contrast to the external queue approach where there is a single external queue, in the integrated approach each node (N_1 to N_5 in Figure 3) maintains its own queues and executes tasks locally. On arrival of a data item I_A, the contacted node forwards this item to the responsible replicas and puts two tasks in its queue. When these tasks are executed, they will write items I_B and I_C. Task execution is distributed within the database.

The basic mechanism of this approach is asynchronous task execution in response to a write to the database. Such a mechanism can be classified as asynchronous triggers in contrast to traditional triggers, which are synchronous. The database acknowledges any write request from the application as soon as the write has finished and the trigger, which implements the corresponding data flow task, has been submit-

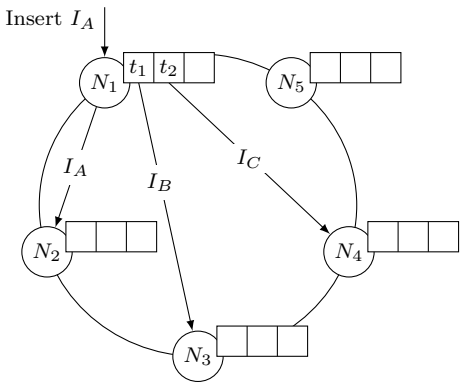


Figure 3: Execution model of the integrated approach: The queue is distributed across nodes of the distributed database, tasks propagate data.

ted for execution into the local queue. The actual execution of the trigger happens in parallel or after the acknowledgement to the application thus guaranteeing low response time to the user. Treating task execution as trigger execution allows us to simplify programming model where the developer specifies a data flow using the well known paradigm of database triggers. Asynchronous execution and fault tolerance mechanisms are transparent to the developer.

As this approach tightly integrates queues into the database, we first specify assumptions about the database that are required to implement this approach.

- The database is partitioned and replicated across nodes according to some replication strategy.
- The application may contact any node of the database to issue write requests. The contacted node serves as the update coordinator. The update coordinator forwards the request to the set of nodes that store the replicas of the record to be modified. The nodes are determined by the key of the modified record according to the replication strategy. The update coordinator waits for responses from all replica nodes. Note that the update coordinator may or may not be included in the set of replica nodes.
- As we discussed in the introduction, the database does not support transactions.
- Each database node is notified (or obtains this information by periodically polling some master node) about failures of any other node.

These assumptions are quite general and exist in practically every modern distributed database and key-value store.

5.1 Programming Model

In the integrated approach, the data flow is defined as a set of triggers. All aspects of asynchronous execution and fault tolerance are transparent to the developer.

To define a trigger, the developer needs to specify the table the trigger is registered on and provide the implementation of the trigger. The trigger implementation is parametrized by (1) the key of a record for which the trigger is fired, (2) the record itself, and (3) the operation that was performed (i.e. insert or delete). Given these parameters the trigger implementation has all the knowledge of the write that fired the trigger. Of course, triggers may also read and write to the database. Triggers should be implemented as idempotent operations: repeated trigger execution should have the same effect as a single execution.

The programming model of this approach has two advantages over the other approaches. First, it is more intuitive for the developer to describe his intentions using a set of triggers, which map well to the application's work flow. In the other approaches the developer has to be much more explicit about how the data flow is implemented and how to deal with failures. Second, the synchronous approach lacks the expressiveness to say that some updates are critical and some others can be done asynchronously. In the external queue approach, these properties have to be programmed out by the developer. In the integrated approach, it is stated explicitly with help of triggers.

5.2 Execution Model

This approach requires changes to the nodes of a database. Each node requires a mechanism to find data flow tasks (instantiations of triggers) that are executed in response to an update; and a queue to buffer tasks that wait to be executed. For each trigger, there is one queue. If there was only one queue for all triggers (and thus, all tasks), we might run into deadlocks for cascading tasks. This will happen if the queue is full and a task execution causes two (or more) different tasks to be executed. Thus, removing one item from the queue but blocking because two items need to be added. Having one queue per trigger prevents those deadlocks. We assume that there are no cycles in the data flow. For each queue, there is a predefined number of worker threads specified as a parameter of the system. Tasks submitted to a queue will be executed by worker threads asynchronously. We designed a master/slave protocol to find and execute tasks. This protocol is able to continue functioning in the presence of node failures.

In our master/slave protocol the update coordinator is considered to be the master. The master is responsible to find and execute data flow tasks. The nodes that are responsible for storing the replicas of the data item are considered slaves. Slaves store backup information about tasks for failure handling. In case the update coordinator is one of the replica nodes it becomes the master only, not a slave, as it makes no sense for the master to store backup information about tasks that it executes.

Protocol of the Master

Data flow tasks are executed at the master for any write request. For each write request, the master is responsible for finding all tasks that need to be executed. The detailed master protocol is shown in Figure 4.

Protocol of a Slave

Slaves maintain a backup task map storing data flow tasks that are executed by the master but have not been reported

On reception of a write request of data item I_A

M_1. Send write messages to slaves and write I_A locally if the master is a replica node

M_2. Wait for acknowledgements from slaves according to the consistency level

M_3. Find tasks for I_A's table $\rightarrow T$

M_4. For each task $t \in T$: submit it into the corresponding queue

M_5. Send acknowledgement to the application

On execution of task t

M_1^*. Execute the task t

M_2^*. Send a notification message containing $hash(t)$ about successful task execution to the slaves. Sending only the hash code of t is sufficient so that slaves can remove backup information of tasks.

Figure 4: Protocols to Find and Execute Data Flow Tasks at the Master.

as completed. This map stores information about tasks using a $hash(task) \rightarrow task$ relation. It makes it easy to remove tasks from the map later. The protocol to backup and remove triggers is shown in Figure 5.

5.3 Failure Model

Failure of the Master

Given the protocol in Figure 4, the failure handling of the master is straight-forward. On reception of a write request, if the master fails before sending the acknowledgement to the application (anywhere before step M_5 in Figure 4), the task might or might not have been executed. Because the application will not get a response, the application retries the write request (at a different node now). Therefore, the trigger will be executed at least once.

If the master fails after having sent the acknowledgement to the application (after step M_5) the task might still not have been executed. The task might wait in the queue at the time the master fails. The application will not retry the write request because the request has been acknowledged. Because the master goes down *after* step M_5, we know that slave nodes have put the task into their backup task maps (because we received their acknowledgements). We assume that the slaves will be notified that the master has failed (see the list of assumptions about the database system above). In that case, each slave will execute all tasks that are stored in the backup task map. It is guaranteed that tasks, which have not been executed by the master, will be executed at-least-once.

In particular, for any task the following holds. If the master fails before executing the task (before step M_1^*), the task is executed at least by the slaves that acknowledged step M_1. If the master fails after the execution but before sending the notifications to the slaves (in between steps M_1^* and M_2^*) the task is executed by the master *and* the slaves. If the

On reception of write message from the master containing data item I_A

S_1. Write I_A locally

S_2. Find tasks for I_A's table $\rightarrow T$

S_3. For each task $t \in T$: put $hash(t), t$ into backup task map

S_4. Send acknowledgement to the master

On reception of notification message containing $hash(t)$

S_1^*. Remove $hash(t)$ from backup task map

Figure 5: Protocols to Backup and Remove Tasks at a Slave.

master fails in the middle of sending notification messages (in the middle of M_2^*) the task is executed by the master and those slaves that did not receive a notification message from the master.

For our protocols we impose fail-stop semantics. When the master recovers from a failure it will start with a clean state. All queues of the execution stages will be empty. All tasks that were stored in these queues are lost. This is acceptable because the slaves executed those tasks.

Failure of a Slave

If a slave fails during any of the steps of the protocol in Figure 5 it will not harm the execution of tasks. The master is able to carry out its protocol despite slave failures. Because we impose fail-stop semantics, when the slave recovers from a failure its backup task map will be empty. All tasks stored in this map are lost. It is possible that a restarted slave receives notification messages without a respective task stored in the map. In that case this (buffered) notification message will be removed after a specific time, which is set programmatically.

5.4 Scalability

As we will show in the experiments, this approach requires less resources to scale that the external queue approach. This is because data flow tasks run on the nodes of the distributed database, which were under-utilized in the external queue approach.

6. EXPERIMENTS AND RESULTS

In order to evaluate the different approaches, we implemented the real-world workload of Twitter. For that we use detailed analyses of Twitter presented in [6] and [7]. In Twitter, whenever a user post a tweet, it is propagated to all followers of this user. That is, the tweet is copied from the user's timeline to all follower timelines. The recent study [7] states that the number of followers per user follows a highly skewed power law distribution: most users have few followers but some users have hundreds of thousands of followers. We used the distribution from [7] to generate the mapping of users to their followers.

	Throughput	Response Time [ms]		
	[msg/s]	median	stddev	max
Synchronous	1297	20.66	286.8	49900
Ext. Queue	1291	0.83	3.46	413
Integrated	1289	1.26	5.17	445

Table 1: Saturating Workload, 2 Database Nodes.

	Throughput	Response Time [ms]		
	[msg/s]	median	stddev	max
Synchronous	1000	5.35	89.21	13800
Ext. Queue	1000	0.84	3.26	141
Integrated	1000	0.80	3.31	171

Table 2: 80% Workload, 2 Database Nodes.

In our experiments, the system is divided into three parts: application, queue, and database. The application acts as load generator and insert tweets into the database at a given rate. The database contains three tables: *tweets*, *followers*, and *timeline*. The tweets table stores all tweets a user has published. The followers table contains the mapping from users to followers. The timeline table contains, for each user, the tweets of himself and of all of its followers. On Twitter, a user usually reads the content of his own timeline. The data flow is the following: each incoming tweet is copied to all follower timelines according to the mapping in the followers table.

We implemented all approaches in Java 1.6 on top of Cassandra [8], a popular distributed database/key-value store used in Twitter, Facebook, Digg, and many other Web applications. Cassandra implements a shared-nothing architecture with data partitioned and replicated across nodes. Cassandra has limited querying capabilities allowing look-ups of records by primary key only, which suffices to build Twitter's data flow. All assumptions required to build the integrated approach listed in Section 5 hold for Cassandra. In particular, Cassandra implements a gossip protocol to notify all nodes about failure of a node. For the synchronous approach and the external queue approach, Cassandra was used without modifications. For the integrated queues approach, we implemented the master and slave protocols described in Section 5 and modified Cassandra's update coordinator code to execute the master protocol each time a data item is written to the tweets table in Cassandra. For all experiments the replication factor in Cassandra was set to 2. The write consistency level was set to *ALL*, the read consistency level to *ONE*. That is, all replicas must be successfully updated to acknowledge a write, but only one node is enough to read from. As external queue, we used Kestrel [12]. Kestrel is the queue implementation used in Twitter [10].

All experiments were carried out using machines with an AMD Opteron 2.4 GHz processor and 6 GB RAM running Ubuntu Server 10. Our experimental setup consisted of four machines. One machine runs the application generating the tweets to be inserted. One machine runs Kestrel, the queue needed in the external queue approach. Two machines run Cassandra, the distributed database. In the scalability experiment, we increase the number of machines up to a total number of 21 machines. In all experiments, we ran the benchmark for 30 minutes. We report on throughput in messages per second and the median, standard deviation, and maximum of response time in milliseconds. One message per second means that the application was able to insert one tweet into the database, query all followers, and propagate the tweet to these followers. The size of each tweet is about 200 bytes. Response time measures the time until the application acknowledges the tweet insertion. For the burst and fault tolerance experiments, we sampled throughput and response time every three seconds and show data collected over a period of three minutes. Furthermore, we collected performance metrics and report on the average CPU load, the sum of main memory used, and sum of incoming and outgoing network traffic (including traffic within the distributed database) of each part of the system.

6.1 Saturating Workload

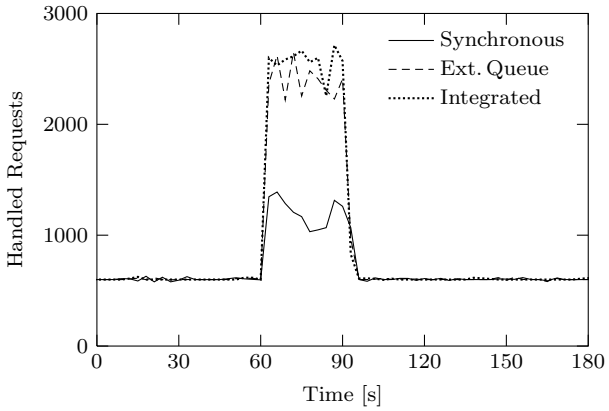
In this experiment we measure the maximum throughput that each approach is able to sustain. The application issues as many tweets as possible to saturate the database. Table 1 shows throughput and response times of the saturating workload. We make the following observations. (1) All approaches peak at a the same maximum throughput of about 1300 messages per second. The limiting factor is the write capacity of the database. It follows that (2) communicating with the external queue does not decrease the maximum throughput the external queue approach. (3) Integrating queues into the database does not decrease the maximum throughput that can be achieved. The additional processing overhead that is shifted from the application to the database is negligible for Twitter's data propagation workload. (4) When the database is saturated, response time cannot be measured accurately as discussed in standard text books [5]. Thus, we will study the response time of a non-saturating workload.

6.2 80% Workload

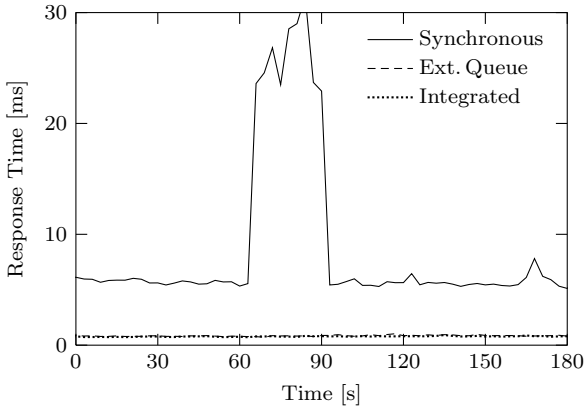
In this experiment we throttle the number of issued tweets to about 80% of the maximum throughput to not saturate the database. The application issues 1000 tweets per second. The results are shown in Table 2. We make the following observations. (1) All approaches handle 1000 messages per second. (2) The median response time of the synchronous approach is about 7 times higher than that of the other approaches. In the synchronous approach the application will acknowledge a tweet only after it has propagated the tweet to all followers. Because of the highly skewed workload (i.e., few users have hundreds of thousands of followers) we observe a high standard deviation and a maximum response time of 13.8 seconds, which is unacceptable. (3) The median response time of the external queue approach and that of the integrated approach are the same. Both asynchronous approaches have a guaranteed maximum response time below 0.2 seconds.

6.3 Bursty Workload

In addition to guaranteed low response time, another advantage of asynchronous execution of data flows is being able

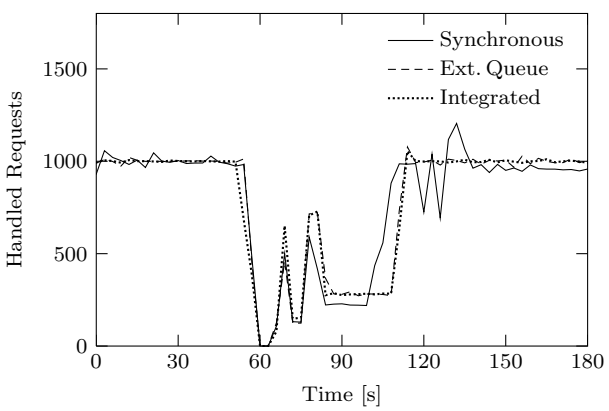


(a) Handled Requests

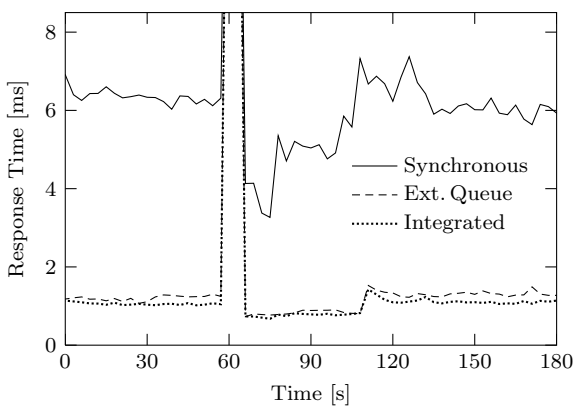


(b) Response Time [ms]

Figure 6: Performance over Time, 30 sec Burst.



(a) Handled Requests



(b) Response Time [ms]

Figure 7: Performance over Time, 30 sec Failure.

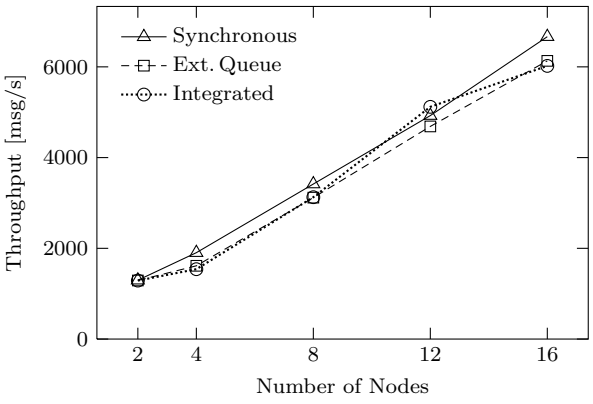
to handle short periods of bursts of requests. Bursts may even exceed the maximum capacity of the database because subsequent operations can be buffered using queues. In this experiment, the rate at which the application issues tweets is the following. First, the application issues tweets at a rate of about 50% of the saturating capacity (600 messages per second). After 60 seconds the request rate jumps to 200% of the saturating capacity (2600 messages per second). This burst lasts for 30 seconds and afterwards the request rate drops back to 50%.

Figure 6(a) shows the number of messages each approach handles over time. We observe that asynchronous approaches can handle short bursts of requests. User tweets can be written at a rate of 2600 messages per second while subsequent writes to forward the tweets to the followers are buffered. The tweets will be forwarded as soon as there is enough capacity (not shown in Figure 6(a)). The synchronous approach is only able to handle requests at the rate of the maximum capacity of 1300 message per second only. It has to drop requests in case of bursts. The asynchronous approaches did not drop messages in this experiment. Figure 6(b) shows the measured response times. For the asynchronous approaches the response time remains constant. For the synchronous approach, the response time jumps from about 6ms to 25ms because the system is saturated during the burst of requests.

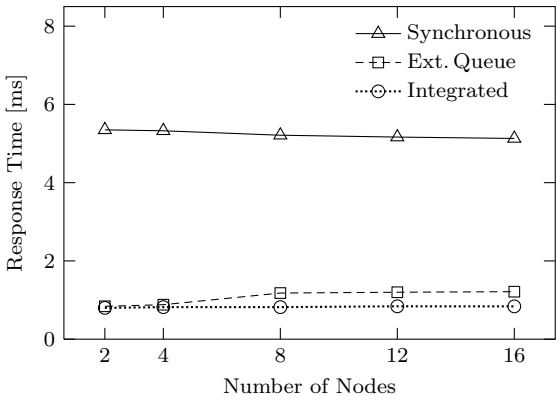
We conclude that (1) asynchronous approaches are able to handle bursts without dropping messages, and (2) the integrated approach shows the same throughput and response time characteristics as the external queue approach.

6.4 Fault Tolerance

In this experiment, we run a database consisting of 4 nodes. The application issues a load of 1000 messages per second. 60 seconds into the experiment, we stop 1 database node by manually killing the database process on this machine. 30 seconds later, we restart the process. Figures 7(a) and 7(b) show throughput and response time for all approaches. For about 5 seconds after the node has faulted, all nodes of the database do not respond. The throughput drops to 0 and requests are stalled until the system responds again. Afterwards, the throughput fluctuates as the database tries to recover from the fault. Throughput stabilizes at around 350 messages per second 20 seconds after the fault. Once we restart the faulted process, it takes the database another 20 to 30 seconds to get back to stable handling of the issued load. Response time of the asynchronous approaches fluctuate less than that of the synchronous approach. In all cases we manually checked that all successfully inserted tweets were also propagated to the respective followers.



(a) Throughput [msg/s], Saturating Workload



(b) Response Time [ms], 80% Workload

Figure 8: Scalability, Vary Database Nodes.

6.5 Scalability

In the last experiment we study the scalability of the different approaches and gather performance metrics. Here, we add nodes to the distributed database and adjust the number of nodes of the application and queue as necessary to obtain best results for each individual approach.

Figure 8(a) shows the throughput of each approach for a saturating workload and varying number of database nodes. All approaches scale equally well as the number of nodes are increased. Figure 8(b) shows the response time of each approach for a 80% workload and varying number of database nodes. The load was set to 80% of the throughput in Figure 8(a). We observe that the response time remains constant as the size of the database is increased.

Tables 3 and 4 show the resource utilization of the saturating experiments for 2 database nodes and 16 database nodes respectively. Here, we make the following observations. (1) The application of the synchronous approach has high CPU load and network traffic because it is responsible for sending all data items across the network to the database. (2) CPU load and network traffic of the application of the external queue approach is even higher than that of the synchronous approach because of the communication with the queue. (3) CPU and network traffic of the integrated approach is lowest compared to the other approaches because the application only sends the session-critical data item to the database. The rest of the data flow is executed within the distributed database, which saves communication overhead and has no noticeable CPU overhead. Main memory usage was not a limiting factor in our experiments. The number of nodes to perform the experiments is lowest for the integrated approach. For 16 database nodes, the synchronous approach needed 3 application nodes to completely saturate the database. The external queue approach needed 3 application nodes (1 load generator and 2 queue workers) and also 2 queue nodes. The integrated approach needed only 2 application nodes. To conclude, when scaling the database to 16 nodes the integrated approach saves 26.2 MB/s in network traffic compared to the external queue approach, which is a reduction of 35%; and requires only a total of 18 nodes instead of 21 nodes, which translates to a cost reduction of about 15%.

7. RELATED WORK

In this paper we focus on propagational data flows, which can be implemented using at-least-once semantics, idempotent writes, and no transactional guarantees. There are other types of data flows out of the scope of our study.

Microsoft SQL Server Service Broker [16] and Oracle Advanced Queuing [9] combine database and queue components to simplify application development and to support more general types of data flows. These types of data flows consist of complex operations requiring strong consistency guarantees. The systems mentioned guarantee exactly-once semantics of individual tasks and support ACID transactions within individual tasks. Because of the requirements of the data flows discussed in this paper, the systems in our study can be implemented without such additional guarantees. Furthermore, the database and the queue in Microsoft SQL Server Service Broker and Oracle Advanced Queuing remain two separate components as in the external queue approach and do not optimize resource utilization.

Google Percolator [11] is a recent example of a system to support analytical data flows. Google Percolator replaced batch-oriented offline web indexing with incremental online web indexing. Web indexing is an analytical data flow described as a series of data aggregation tasks. For these types of data flows at-most-once semantics of task execution is sufficient because skipping some intermediate data items still leads to statistically correct results, for example computing PageRank [2]. Still, each task needs ACID transactions to synchronize modifications of aggregate values. Google Percolator provides at-most-once semantics of task execution and ACID transactions within individual tasks.

To the best of our knowledge there is no scientific study integrating queues and databases for propagational data flows.

There is a great deal of work on *synchronous* triggers inside database systems [3, 13, 15]. We are not aware of any proposals of triggers that are executed asynchronously.

8. CONCLUSION

In this paper we studied various architectures to build distributed data flows using distributed key-value stores. We

	CPU [%]			Memory [GB]			Network Traffic [MB/s]			Number of Nodes			
	App	DB	Queue	App	DB	Queue	App	DB	Queue	App	DB	Queue	Total
Synchronous	11.0	39.6	–	1.76	2.98	–	2.06	8.90	–	1	2	–	3
External Queue	21.6	36.4	16.6	1.98	3.94	0.83	2.87	8.83	0.68	1	2	1	4
Integrated	2.30	40.9	–	1.72	3.85	–	0.36	7.41	–	1	2	–	3

Table 3: Resource Utilization, Saturating Workload, 2 Database Nodes.

	CPU [%]			Memory [GB]			Network Traffic [MB/s]			Number of Nodes			
	App	DB	Queue	App	DB	Queue	App	DB	Queue	App	DB	Queue	Total
Synchronous	22.1	41.9	–	8.64	24.1	–	13.6	57.7	–	3	16	–	19
External Queue	38.8	41.9	33.2	7.03	23.7	1.73	16.0	54.0	4.06	3	16	2	21
Integrated	6.17	41.1	–	4.77	25.9	–	2.34	45.6	–	2	16	–	18

Table 4: Resource Utilization, Saturating Workload, 16 Database Nodes.

showed that asynchronous mechanisms are key when building data flows as it allows to guarantee low response time in presence of skewed workloads and to handle short periods of bursts. Our experiments confirm that the state-of-the-art approach based on an external queue provides desirable performance characteristics but does not utilize resources efficiently.

We addressed this issue by integrating queues into the key-value store, which allows to utilize resources more efficiently and, thus, scale the system with fewer nodes. We also showed that this integrated approach has the following additional advantages: (1) an easier programming model, which encapsulates all the details of asynchronous execution and failure handling; and (2) a tight integration into the database, which eliminates maintenance and administration cost for additional products such as an external queue.

We have proposed the key mechanisms of the integrated approach, asynchronous trigger execution and protocols for fault tolerance, as an extension to the next version of the Cassandra distributed database (see Cassandra Ticket 1311, https://issues.apache.org/jira/browse/cassandra-1311).

9. REFERENCES

[1] A. Avram. Twitter, an evolving architecture. http://www.infoq.com/news/2009/06/Twitter-Architecture, 2009.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1–7):107–117, 1998.

[3] K. P. Eswaran. Aspects of a trigger subsystem in an integrated database system. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 243–250, San Francisco, CA, USA, 1976.

[4] P. Helland and D. Campbell. Building on quicksand. In *Proceedings of the 4th Conference on Innovative Data Systems Research*, CIDR '09, Asilomar, CA, USA, 2009.

[5] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley, New York, NY, USA, 1991.

[6] B. Krishnamurthy, P. Gill, and M. Arlitt. A few chirps about Twitter. In *Proceedings of the 1st Workshop on Online Social Networks*, WOSN '08, pages 19–24, Seattle, WA, USA, 2008.

[7] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, Raleigh, NC, USA, 2010.

[8] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[9] Oracle streams advanced queuing user's guide. http://www.sysdba.de/oracle-dokumentation/11.1/server.111/b28420.pdf, 2007.

[10] G. Pass. Building on open source. *Twitter Blog*, http://blog.twitter.com/2009/01/building-on-open-source.html, 2009.

[11] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 251–264, Vancouver, BC, Canada, 2010.

[12] R. Pointer. Kestrel. http://github.com/robey/kestrel, 2011.

[13] S. Potamianos and M. Stonebraker. *The POSTGRES Rules System*. Morgan Kaufmann, San Mateo, CA, USA, 1996.

[14] B. Stone. Inauguration day on Twitter. *Twitter Blog*, http://blog.twitter.com/2009/01/inauguration-day-on-twitter.html, 2009.

[15] J. Widom. The starburst active database rule system. *IEEE Transactions On Knowledge and Data Engineering*, 8:583–595, August 1996.

[16] R. Wolter. Building reliable, asynchronous database applications using Service Broker. http://msdn.microsoft.com/en-us/library/ms345113(SQL.90).aspx, 2005.