Taking Omid to the Clouds: Fast, Scalable Transactions for Real-Time Analytics

Ohad Shacham Yonatan Gottesman Edward Bortnikov Eshcar Hillel
Yahoo Research, Oath

Idit Keidar
Yahoo Research, Oath and Technion

Abstract

Omid is a transaction processing system for Apache HBase, originally designed for data processing pipelines at Yahoo, which are, by and large, throughput-oriented monolithic NoSQL applications. Providing a platform to support converged real-time transaction processing and analytics applications – dubbed *translytics* – introduces new functional and performance requirements. For example, SQL support is key for developer productivity, multi-tenancy is essential for cloud deployment, and latency is cardinal for just-in-time data ingestion and analytics insights.

We discuss our efforts to adapt Omid to these new domains, as part of the process of integrating it into Phoenix as the transaction processing backend. A central piece of our work is latency reduction in Omid's protocol, which also improves scalability. Under light load, the new protocol's latency is 4x to 5x smaller than the legacy Omid's, whereas under increased loads it is an order of magnitude faster.

1 Introduction

In recent years, transaction processing technologies have paved their way into multi-petabyte big data platforms [7, 5, 9]. Modern industrial *transaction processing systems (TPSs)* [7, 9, 2, 1] complement existing underlying NoSQL key-value storage with *atomicity*, *consistency*, *isolation* and *durability* (ACID) semantics that enable programmers to perform complex data manipulation without overcomplicating their applications.

Such technologies must evolve in light of the recent paradigm shift towards massive deployment of big data services in public clouds; for example, the AWS cloud can run HBase as part of Elastic MapReduce, and SQL engines like Apache Phoenix increasingly target public cloud deployment. Yet adapting TPS technology for cloud use is not without challenges, as we now highlight.

1.1 Challenges

Diverse functionality. Transaction support was initially motivated by specific use cases like content indexing for web search [7, 9] but rapidly evolved into a wealth of OLTP and analytics applications (e.g., [10]). Today, users expect to manage diverse workloads within a single data platform, to avoid lengthy and error-prone extract-transform-load processes. A Forrester report [3] coins the notion of *translytics* as "a unified and integrated data platform that supports multi-workloads such as transactional, operational, and analytical simultaneously in real-time, ... and ensures full transactional integrity and data consistency".

As use cases become more complex, application developers tend to prefer high-level SQL abstractions to crude NoSQL data access methods. Indeed, scalable data management platforms (e.g., Google Spanner [5], Apache Phoenix, and CockroachDB [1]) now provide full-fledged SQL interfaces allowing complex query semantics with strong data guarantees. SQL APIs raise new requirements for transaction management, e.g., in the context of maintaining secondary indexes for accelerated analytics.

Scale. Public clouds are built to serve a multitude of applications with elastic resource demands, and their efficiency is achieved through scale. Thus, cloud-first data platforms are designed to scale well beyond the limits of a single application. For example, Phoenix is designed to scale to 10K query processing nodes in one instance, and is expected to process hundreds of thousands or even millions of transactions per second (*tps*).

Latency. Whereas early applications of big data transaction processing systems were mostly throughput-sensitive [7, 9], with the thrust into new interactive domains like social networks, messaging and algorithmic trading latency becomes essential. SLAs for interactive user experience mandate that simple updates and point queries complete within single-digit milliseconds. The early experience with Phoenix shows that programmers often refrain from using transaction semantics altogether for fear of high latency, and instead adopt solutions that may compromise consistency.

Multi-tenancy. Data privacy is key in systems that host data owned by multiple applications. Maintaining access rights is therefore an important design consideration for TPSs, e.g., in maintaining shared persistent state.

1.2 Evolving Omid for public clouds

This paper describes recent functionality extensions and performance improvements we made in Omid – an open source transaction manager for HBase – to power Phoenix[1], a Hadoop SQL-compliant data analytics platform designed for cloud deployment.

In contrast to earlier SQL engines for Hadoop (e.g., Hive and Impala), which focused on immutable data, Phoenix targets converged data ingestion and analytics [4], which necessitates transactions, and its applications are both latency and throughput sensitive. Phoenix uses HBase as its key-value storage layer, and Omid for transaction processing[2].

This paper describes two concrete contributions to Omid and Phoenix. First, Section 3 describes the protocol's redesign for low-latency. The new protocol, *Omid Low Latency (Omid LL)*, dissipates Omid's major architectural bottleneck. It reduces the latency of short transactions by 5x under light load, and by 10x–100x under heavy load. It also scales the overall system throughput to 550K tps while remaining within real-time latency SLAs. In contrast to previously published protocols (e.g., [7]), our solution is amenable to multi-tenancy. The development of this new feature is reported in https://issues.apache.org/jira/browse/OMID-90.

Second, Section 4 discusses our *SQL compliance*. We add support for creating secondary indexes on-demand, without impeding concurrent database operations or sacrificing consistency. We further extend the traditional SI model, which provides single-read-point-single-write-point semantics, with multiple read and write points. This functionality is used to avoid recursive read-your-own-writes scenarios in complex queries. The development of these new features in Omid and Phoenix is reported in https://issues.apache.org/jira/browse/OMID-82 and https://issues.apache.org/jira/browse/PHOENIX-3623, respectively.

2 API and Context

Omid is a transaction processing system that runs atop an underlying key-value store such as HBase and allows bundling multiple data operations into a single atomic transaction.

API and semantics. The data store holds *objects* (often referred to as *rows* or *records*) identified by unique *keys*. Each row can consist of multiple *fields*, representing different *columns*. We consider multi-versioned objects, where object values are associated with *version numbers*, and multiple versions associated with the same key may co-exist. The data store provides the following API calls: get, scan, put, remove, and check&mutate.

[1] https://phoenix.apache.org

[2] Tephra is supported as alternative transaction manager, though its scalability and reliability are inferior to Omid's.

TPSs provide *begin* and *commit* APIs for delineating transactions: a *transaction* is a sequence of *read* and *write* operations on different objects that occur between begin and commit. For simplicity, we only describe single-key (get and put) operations here; multi-key range queries (scans) are treated as sequences of reads.

A TPS ensures the ACID properties for transactions: *atomicity* (all-or-nothing), *consistency* (preserving each object's semantics), *isolation* (in that concurrent transactions do not see each other's partial updates), and *durability* (whereby updates survive crashes).

Different isolation levels can be considered for the third property. Omid, like other popular DBMSs and TPSs (e.g., Oracle, PostgreSQL, and SQL Server, Percolator, and CockroachDB) satisfies SI. Intuitively, this means that every transaction sees a consistent "snapshot" of the database. For example, if a transaction updates the values of two stocks, then no transaction may observe the old value of one of them and the new value of the other. Under SI, two concurrent transactions conflict only if they both update the same item.

Following a commit call, the transaction may successfully *commit*, whereby all of its operations take effect, or *abort*, in which case none of its changes take effect.

Big data platforms. Apache HBase is one of the most scalable key-value storage technologies available today. Phoenix complements the HBase storage tier with a query processing (compute) tier. The latter scales independently (the current scalability goal is 10,000 query servers). Phoenix compiles every SQL statement into a plan, and executes it on one or more servers. Its query processing code invokes the underlying HBase for low-level data access, and a TPS (Omid or Tephra) for transaction management.

Wherever possible, Phoenix strives to push computation close to data (e.g., for filtering and aggregation), in order to minimize cross-tier communication. For this, it makes extensive use of server-side stored procedures, which in HBase are supported by the non-intrusive *coprocessor* mechanism.

3 Low-Latency Transactions

We overview Omid LL's design choices in Section 3.1 and give the protocol in Section 3.2. Section 3.3 presents exemplary performance results.

3.1 Omid LL design choices

Our design choices are geared towards high performance without sacrificing cloud deployability. Table 1 compares our choices with ones made in other TPSs.

Centralized validation. Snapshot isolation requires validating write-sets – i.e., checking for write-write conflicts – upon commit. Omid LL adopts Omid's centralized conflict

TPS	validation	commit entry updates	multi tenancy
Percolator	**D**	**D**	no
CockroachDB	**D**	**D**	yes
Omid1, Tephra	**C**	**R**	yes
Omid	**C**	**C**	yes
Omid LL	**C**	**D**	**yes**

Table 1: Design choices in TPSs. **C** – centralized, **D** – distributed, **R** – replicated.

detection mechanism, which eliminates the need for locking objects in the data store, and is extremely scalable [9].

Other TPSs (like Percolator and CockroachDB [1]) instead use a distributed 2PC-like protocol that locks all written objects during validation (either at commit time or during the write). To this end, they use atomic check&mutate operations on the underlying data store. This slows down either commits (in case of commit-time validation) or transactional writes (in case of write-time validation), which takes a toll on long transactions, where validation time is substantial.

Distributed commit entry. Because a transaction may write many keys and must commit them all in one atomic data store operation, TPSs keep per-transaction *commit entries*, which are the source of truth regarding the transaction status (pending, committed, or aborted). The early generation of Omid [6] (referred to as Omid1) and Tephra replicate commit entries of pending transactions among all active clients, which consumes high bandwidth and does not scale. Omid instead uses a dedicated *commit table (CT)*, and has the centralized TM persist all commits to this table.

Our experiments show that the centralized access to commit entries is Omid's main scalability bottleneck, and while this bottleneck is mitigated via batching, this also increases latency. Omid chose this option as it was designed for high throughput. Here, on the other hand, we target low latency. We therefore distribute the commit entry updates, and allow commit entries of different transactions to be updated in parallel by independent clients. We further shard this table evenly across many nodes in order to spread the load. The distribution of commit entry updates reduces latency by up to an order of magnitude for small transactions.

Multi-tenancy. We note that commit table updates are distributed also in CockroachDB and Percolator. The latter takes this approach one step further, and distributes not only the commit table updates but also the actual commit entries. There, commit entries reside in user data tables. The problem with this approach is that it assumes that all clients have permissions to access all tables. For example, a transaction attempting to read from table A may encounter a tentative (non-committed) write produced by a transaction that accessed table B before table A, and will need to refer to that transaction's commit entry in table B in order to determine its status. This approach did not pose a problem in Percolator, which was designed for use in a single application (Google Search), but is unacceptable in multi-tenant settings.

Unlike data tables, which are owned by individual applications that manage their permissions, the dedicated commit table is owned by the TPS; it is accessed exclusively by the TPS client library, which in turn is only invoked internally by the database engine, and not by application code.

3.2 Omid LL algorithm

Algorithm 1 Omid LL's TM algorithm.

```
1:  procedure BEGIN
2:      checkRenew()      ▷ check and renew lease for HA support
3:      return Clock.fetchAndIncrement()
4:  procedure COMMIT(txid, write-set)
5:      checkRenew()      ▷ check and renew lease for HA support
6:      ts_c ← Clock.fetchAndIncrement()
7:      if conflictDetect(txid, write-set, ts_c) then
8:          return ts_c
9:      else
10:         return ABORT
```

When a transaction begins, it obtains a read timestamp (version) ts_r for reading its consistent snapshot. To commit, it obtains a commit timestamp, ts_c and persists a commit entry in the CT that indicates that the transaction with read timestamp ts_r has committed with timestamp ts_c.

Whereas Omid's CT is updated by the TM, Omid LL distributes the CT updates amongst the clients. Its TM is thus a simplified version of Omid's TM, and appears in Algorithm 1. It has two roles: First, it allocates read and commit timestamps by fetching-and-incrementing a monotonically increasing global clock. Second, upon commit, it calls the *conflictDetect* function, which performs validation using an in-memory hash table by checking, for each key in the write-set, that its commit timestamp in the hash-table is smaller than the committing transaction's ts_r. If there are no conflicts, it updates the hash table with the write-set of the new transaction and its ts_c. The *checkRenew* procedure supports the TM's high availability (HA), and its description is beyond the scope of this paper; see [9, 8] for details.
Client operations proceed as follows (cf. Algorithm 2):

Begin. The client sends a begin request to the TM, which assigns it a read timestamp ts_r.

Write. During a transaction, a write operation *tentatively* writes a new value to an object with a certain version number. The version is the transaction's ts_r, which exceeds all versions written by transactions that committed before it began. A dedicated *commit* column is initialized to nil to indicate that the write is still tentative (not committed); after the transaction commits successfully, the value of the commit field is set to the writing transaction's commit timestamp. The write further tracks the key in its local write-set.

Algorithm 2 Omid LL's client-side operations.

```
procedure BEGIN
    return TM.begin
procedure WRITE(ts_r, key, fields, values)
    track key in write-set
    add commit to fields and nil to values
    return ds.put(key, ts_r, fields, values)
procedure READ(ts_r, key)
    for rec ← ds.get(key, versions down from ts_r) do
            ▷ set ts_c to the commit timestamp associated with rec
        if rec.commit ≠nil then                    ▷ commit cell exists
            ts_c ← rec.commit
        else                                ▷ commit cell is empty, check CT
            ts_c ← CT.get(rec.ts)
            if ts_c = nil then              ▷ no CT entry, set status to ⊥
                ts_c ←CT.check&mutate(rec.ts, commit, nil, ⊥)
                if ts_c=nil then            ▷ forced abort successful
                    ts_c ← ⊥
            if ts_c = ⊥ then
                ▷ check for race: commit before ⊥ entry created
                re-read rec from ds
                if rec.commit ≠nil then
                    ts_c ← rec.commit
                    CT.remove(rec.ts)
                else continue               ▷ writing transaction aborted
        if ts_c < ts_r then
            return rec.value
    return nil                      ▷ no returnable version found in loop
procedure COMMIT(ts_r, write-set)
    ts_c ← TM.commit(ts_r, write-set)          ▷ may return abort
    if ts_c ≠ abort then
        if CT.check&mutate(ts_r, commit, nil, ts_c) = ⊥ then
            ts_c ← abort
    for all keys k ∈ write-set do
        if ts_c = abort then ds.remove(k, ts_r)
        else ds.put(k, ts_r, commit, ts_c)
    CT.remove(ts_r)                        ▷ garbage-collect CT entry
```

Read. The algorithm traverses data records (using the data store's ds.get API) pertaining to the requested key with a version that does not exceed ts_r, latest to earliest, and returns the first value that is committed with a version smaller than or equal to ts_r. To this end, it needs to discover the commit timestamp, ts_c, associated with each data record it considers.

If the record is tentative (commit=nil), then we do not know whether the transaction that wrote it is still pending. Since read cannot return without determining its final commit timestamp, it must forcefully abort it in case it is still pending, and otherwise discover its outcome. To this end, read first refers to the CT. If there is no CT entry associated with the transaction (ts_c=nil), read attempts to create an entry with $\perp$ as the commit timestamp. This is done using an atomic check&mutate operation, due to a possible race with a commit operation.

There is another subtle race to consider in case the commit record is set to $\perp$ by a read operation. Consider a slow read operation that reads the data record rec when its writing transaction is still pending, and then stalls for a while. In the interim, the writing transaction successfully commits, updates the data record, and finally garbage-collects its CT entry. At this point the slow reader wakes up and does not find the commit entry in the CT. It then creates a $\perp$ entry even though the transaction had successfully committed. To discover this race, we have a read operation that either creates or finds a $\perp$ entry in the CT re-read the data record.

Commit. The client first sends a commit request to the TM. If the TM detects conflicts then the transaction aborts, and otherwise the TM provides the transaction with its commit timestamp ts_c. The client then proceeds to commit the transaction, provided that no read had forcefully aborted it. To ensure the latter, the client uses an atomic check&mutate to create the commit entry.

To avoid an extra read of the CT on every transactional read, once a transaction is committed, it writes the transaction's ts_c to the commit columns of all keys in the transaction's write-set. In case of an abort, it removes the pending records. Finally, it removes the transaction's CT entry.

3.3 Performance evaluation

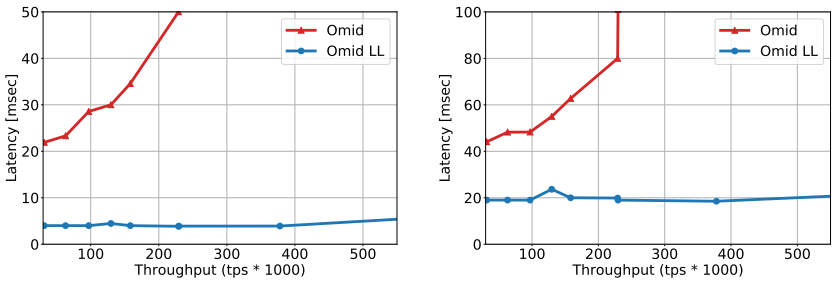
We compare Omid LL's performance to that of Omid. Our experiment testbed consists of nine 12-core Intel Xeon 5 machines with 46GB RAM and 4TB SSD storage, interconnected by 10G Ethernet. We allocate three of these to HBase nodes, one to the TM, one to emulate the client whose performance we measure, and four more to simulate background traffic. Our test cluster stores approximately 23M keys (∼7M keys per node). The values are 2K big, yielding roughly 46GB of actual data, replicated three-way in HDFS. The keys are hash-partitioned across the servers. The data accesses are 50% reads and 50% writes. The key access frequencies follow a Zipf distribution

Figure 1 presents the average latency of single-key and ten-operation transactions as a function of system throughput. Additional experiments are reported in [8].

4 SQL Oriented Features

Comprehensive SQL support in Phoenix involved functionality extensions as well as performance optimizations.

Performance-wise, Phoenix extensively employs stored procedures implemented as HBase coprocessors in order to eliminate the overhead of multiple round-trips to the data store. We integrated Omid's code within such HBase-resident procedures. For example, Phoenix coprocessors invoke transactional reads, and so we implemented Omid's transactional read – the loop implementing read in Algorithm 2 – as a coprocessor as well. This allowed for a smooth integration with Phoenix and also reduced the overhead of transactional reads when multiple versions are present.



(a) Transaction size = 1

(b) Transaction size=10

Figure 1: Latency vs. throughput in Omid and Omid LL.

Functionality-wise, we added support for on-the-fly construction of secondary indexes (Section 4.1) and extended Omid's SI model to allow multiple read- and write-points, as required in some Phoenix applications (Section 4.2).

4.1 Index construction

A secondary index in SQL is an auxiliary table that provides fast access to data in a table by a key that is different from the table's primary key. This need often emerges in analytics scenarios, in which data is accessed by multiple dimensions. Typically, the secondary key serves as the primary key of the secondary index, and is associated with a unique reference into the base table (e.g., primary key + timestamp). SQL query optimizers exploit secondary indexes in order to produce efficient query execution plans. Query speed is therefore at odds with update speed since every write to a table triggers writes to all its indexes.

The SQL standard allows creating indexes on demand. When a user issues a CREATE INDEX command, the database (1) populates the new index table with data from the base table, and (2) installs a trigger to augment every new write to the base table with a write to the index table.

We exploit Omid's SI semantics in order to create such indexes. To achieve (1), Phoenix invokes a transaction that scans a snapshot of the base table and streams the data into the index table, without blocking concurrent puts. Once the bulk population completes, the index can become available to queries. To achieve (2), the database creates a trigger that augments all transactions that update the base table with an additional update of the secondary index.

In order to guarantee the new index's consistency with respect to the base table, the snapshot creation and the trigger setup must be atomic. In other words, all writes beyond the snapshot timestamp must be handled by the trigger. Omid achieves this through a new *fence* API implemented by the TM, which is invoked when the trigger is installed. A fence call produces a new *fence timestamp* by fetching-and-incrementing the TM's clock, and records the fence timestamp in the TM's local state. Subsequently, the TM aborts every transaction whose read timestamp precedes the fence's timestamp and attempts to commit after it. Note that there is at most one fence timestamp per index at a given time.

Neither the bulk index population nor its incremental update require write conflict detection among the index keys, for different reasons. The former does not contend with any other transaction, and hence is committed automatically – the commit cells are created simultaneously with the rest of the index data. The latter is voided by the TM detecting conflicts at the base table upon commit. Hence, in transactions augmented with secondary index updates, there is no need to add the secondary index keys to the transaction's write-set, and so the load on the TM does not increase. Omid extends its put API for this scenario.

4.2 Extended snapshot semantics

Some Phoenix applications require semantics that deviate from the standard SI model in that a transaction does not see (all of) its own writes. We give two examples.

Checkpoints and snapshots. Consider a social networking application that stores its adjacency graph as a table of neighbor pairs. The transitive closure of this graph is computed in multiple iterations. Each iteration scans the table, computes new edges to add to the graph, and inserts them back into the table. Other functions like PageRank are implemented similarly. Such programs give rise to the pattern given in Figure 2.

Note that the SQL statement loops over entries of T (in other words, the program is a nested loop). Normally, the SI model enforces read-your-own-writes semantics. However, in this case the desirable semantics are that the reads only see data that existed prior to the current statement's execution. This isolation level is called *snapshot isolation exclude-current (SIX)*.

To support this behavior, we implement in Omid two new methods, *snapshot* and *checkpoint*. Following a checkpoint, we move to the SIX level, where reads see the old snapshot (i.e., updates that precede the checkpoint call) and writes occur in the next snapshot, which is not yet visible. The snapshot call essentially resets the semantics back to SI, making all previous writes visible. Given these two calls, Phoenix translates SQL statements as above to the loop in Figure 2– it precedes the evaluation of the nested SQL statement with a

SQL program that requires SIX isolation:

```
for iterations i = 1, ... do
    insert into T as select func(T.rec) from T where ...
```

Calls generated by Phoenix for the SQL program:

```
for iterations i = 1, ... do
    checkpoint                                  ▷ move to SIX isolation
    iterator ← T.scan(...)                      ▷ scan in current snapshot
    for all rec ∈ iterator.getNext() do         ▷ parallel loop
        T.put(func(rec))                        ▷ write to next snapshot
    snapshot                                    ▷ make all previous writes visible
```

Figure 2: SQL program requiring SIX isolation and corresponding Phoenix execution plan.

checkpoint, and follows it with a snapshot. Thus, each SQL statement sees the updates of the preceding iteration but not of the current one.

To support the SIX isolation level in Omid, the TM promotes its transaction timestamp counter ts_r in leaps of some $\Delta > 1$ (rather than 1 as in the legacy system). The client manages two distinct local timestamps, τ_r for reads and τ_w for writes. By default, it uses $\tau_r = \tau_w = ts_r$, which achieves the traditional SI behavior. Omid then has the new methods, snapshot and checkpoint, increment τ_r and τ_w, respectively. If the consistency level is set to SIX, it maintains $\tau_w = \tau_r + 1 < ts_r + \Delta$, thereby separating the reads from the writes. A transaction can generate $\Delta - 1$ snapshots without compromising correctness. By default, Omid uses $\Delta = 50$.

Snapshot-all. When a transactions involves multiple snapshots that update the same key, it is sometimes required to read *all* versions of key that existed during the transaction. This is called the *snapshot-all* isolation level.

We use this isolation level, for example, when aborting a transaction that involves multiple updates to a key indexed by a secondary index. Consider a key k that is initially associated with a value v_0, and a secondary index maps v_0 to k. When a transaction updates k to take value v_1, a tentative deletion marker is added to v_0 in the secondary index while v_1 is added. If the transaction then updates the same key (in an ensuing snapshot) to take the value v_2, then v_1 is marked as deleted and v_2 is added, and so on. In case the transaction aborts, we need to roll-back all deletions. But recall that index updates are not tracked in the write-set, and so we need to query the data store in order to discover the set of updates to roll back. We do this by querying k in the primary table with the snapshot-all isolation level, to obtain all values that were associated with k throughout the transaction, beginning with v_0. We can then clean up the redundant entries in the secondary index, and remove the deletion marker from v_0.

5 Conclusion

As transaction processing services begin to be used in new application domains, low transaction latency and rich SQL semantics become important considerations. In addition, public cloud deployments necessitate solutions compatible with multi-tenancy. Motivated by such use cases we evolved Omid for cloud use. As part of this evolution, we improved Omid's protocol to reduce latency (by up to an order of magnitude) and improve throughput scalability. We have further extended Omid with functionalities required in SQL engines, namely secondary index construction and multi-snapshot semantics. We have successfully integrated it into the Apache Phoenix translytics engine.

Acknowledgments

We thank Aran Bergman and James Taylor for fruitful discussions.

References

[1] CockroachDB. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[2] Tephra: Transactions for Apache HBase. https://tephra.io.

[3] The Forrester Wave: Translytical Data Platforms, Q4 2017. https://reprints.forrester.com/#/assets/2/364/RES134282/reports.

[4] Who is using Phoenix? https://phoenix.apache.org/who_is_using.html.

[5] J. C. Corbett and et al. Spanner: Google's globally-distributed database. In *OSDI 12*, 2012.

[6] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *ICDE 14*.

[7] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 10*, 2010.

[8] O. Shacham, Y. Gottesman, A. Bergman, E. Bortnikov, E. Hillel, and I. Keidar. Taking omid to the clouds: Fast, scalable transactions for real-time cloud analytics. In *VLDB 18, Industry Track*, 2018.

[9] O. Shacham, F. Perez-Sorrosal, E. Bortnikov, E. Hillel, I. Keidar, I. Kelly, M. Morel, and S. Paranjpye. Omid, reloaded: Scalable and highly-available transaction processing. In *FAST 17*, 2017.

[10] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littleeld, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. In *VLDB*, 2013.