Towards Generic Fine-Grained Transaction Isolation in Polystores

Nuno Faria[0000−0003−4691−0440], José Pereira[0000−0002−3341−9217], Ana Nunes Alonso[0000−0002−0519−9675], and Ricardo Vilaça[0000−0002−6957−1536]

INESC TEC and U. Minho, Portugal
{nuno.f.faria,jose.o.pereira,ana.n.alonso,ricardo.p.vilaca}@inesctec.pt

Abstract. Transactional isolation is a challenge for polystores, as along with the limited capabilities of each datastore, we have to contend with their sheer diversity. However, transactional isolation is increasingly desirable as a variety of datastores are being sought after for roles that go beyond simple data lakes, where information is mostly static. Transactional guarantees are also relevant for reliability at scale. Finally, it would also close the gap to what is available in multi-model database systems. In this paper, we propose that transactional isolation in polystores can be achieved by leveraging the query engine, i.e., we implement some of the responsibilities of a transactional storage manager (TSM) in the query language itself. This has the key advantage of greatly simplifying design and implementation, as it doesn't need to be re-invented for each datastore, and should increase performance, by taking advantage of dynamic query optimization where available. We demonstrate the feasibility of the proposal with a simple proof-of-concept and experiment.

Keywords: transactions· snapshot isolation· polystores

1 Introduction

Polystores aim at combining the diversity of data models, query languages and interfaces, and architectures of multiple datastores [17, 22]. Although these target, mainly, big data and analytical workloads, transactional updates are desired for reasons such as the ability to correct and update data, consistency considering concurrent updates, and ultimately for reliability, even if not handling frequent updates [7]. More recently, this need arises with fast-changing data incoming from different sources, as is the case with IoT sensors and periodic synchronization with edge systems [13].

Traditionally, transactional isolation and recovery are the responsibility of the transactional storage manager layer [21]. Depending on the strategy used, these are achieved by the combined effect of the lock manager, the buffer pool (i.e., for latching and holding different versions), and the log manager. These features are implemented separately and lie beneath the query engine, which then operates within the abstraction of an isolated and recoverable data space. More recently,

transactional isolation has also been provided for NoSQL datastores as a custom middleware layer that wraps the native store [19].

Unfortunately, transactional isolation in polystores is harder than in traditional database systems or homogeneous big datastores, and often identified as a key research challenge [28, 30]. The first issue is that target datastores have wildly different isolation and consistency criteria, and not just different implementations of similar criteria. Namely, some systems, such as MongoDB [26] or Neo4j [5], provide multi-operation isolation and recovery. Other systems, such as HBase, do not offer multi-operation isolation but provide multi-versioning and a re-do log, that can be used for transactional isolation at the middleware level [19]. Still, some systems (e.g., Cassandra [23]) exhibit no isolation at all and offer only eventual consistency [31], which is central to their value proposition as distributed and scalable. The second issue is how to enforce a single transactional context for an operation reading from or updating multiple datastores. Even datastores that have transactional support such as MongoDB or Neo4j do not support XA [1] transaction interfaces for two-phase commit. Therefore, individually wrapping or modifying each datastore with a transactional storage management layer is both unfeasible and undesirable.

In this paper, we assume Snapshot Isolation [9] as the target transactional isolation criterion and the use of a multi-version optimistic concurrency control mechanism. We divide transactional processing into handling two overarching concerns: The first involves capturing write operations and, when the commit is requested, validating that there are no write-write conflicts with concurrent transactions; the second is the ability to, at any point during the execution of a given transaction, reconstruct the current snapshot by reconciling values written by previously committed transactions, items updated by the current transaction and avoiding values written by concurrent transactions. We address only the latter and focus on the computation needed to deliver the snapshot in a polyglot query engine.

Our first requirement is to provide transactional isolation and recovery, while at the same time allowing unfettered access to native stores. This precludes, for instance, cluttering the data with version information. The second requirement stems from the observation that the best approach for computing isolated snapshots varies for different datastores and that an efficient implementation must take advantage of each one's strengths.

The main insights in this paper are that reconstructing a transactional snapshot across a diversity of datastores (1) is itself a polyglot data processing problem and (2) that we can take advantage of an optimizing query engine to make it simpler, portable, and efficient. We are, as the saying goes, "eating our own dog-food."

2 Background and assumptions

2.1 Query processing

We assume as the baseline a cloud multi-datastore query engine such as CloudMdsQL [22] offering a SQL-like language that can embed statements in native query languages of diverse datastores as table expressions, i.e., *native table expressions*. It follows the mediator/wrapper architecture from multi-database systems: A logically centralized mediator handles client connections, parses and optimizes queries, and then hands sub-sets of the resulting plan for each target datastore to each wrapper, that extracts native query fragments or converts relational operators in the plan and handles execution and data transfer.

In practice, this means that ad-hoc views of data from multiple datastores can be defined and used in relational queries. A relational data model, extended with non-atomic list and dictionary types, is used as the target for such views and the domain for queries in the CloudMdsQL common query language. The major advantage of this approach is that it is able to fully exploit the power of each datastore with native queries without having to fully map data to a common data model, while at the same time globally optimizing the composite query, e.g. by pushing down selection predicates, using bind join, performing join ordering, or planning intermediate data shipping.

2.2 Versions and snapshot isolation

We assume Snapshot Isolation [9] as the target criterion. In contrast to traditional ANSI isolation levels based on 2-phase locking, using a multi-version concurrency control mechanism has clear advantages for read-only transactions and parallel/distributed systems, and is now widely preferred.

This means that there can be multiple versions of each data item stored at the same time and that a version is visible to a transaction if and only if it had been committed before the transaction started. For simplicity, we consider only full Snapshot Isolation, with multi-statement consistency, and not the weaker single-statement Read Consistency levels that are also available in various systems.

Assuming that the minimum visibility (commit) timestamp for an item is cts and that the maximum (starting) visibility timestamp for a transaction is sts, we can consider these possible states for each usable version of an item:

Visible-to-All (or Storage): Committed versions labeled with a cts that is less than or equal to the starting timestamps sts of all currently executing transactions, thus, visible-to-all transactions unless overwritten.

Visible-to-Some (or Cache): Committed versions labeled with a cts that is greater than the starting timestamp sts of some currently executing transactions, thus, invisible to such transactions even if not overridden. Keeping these versions separate from those visible-to-all avoids non-repeatable reads.

Visible-to-One (or Temporary): Uncommitted versions associated with a single transaction. These versions ensure that a transaction reads its own

writes and at the same time avoid causing dirty reads in concurrent transactions.

When a version is written, it starts in the visible-to-one state, proceeds to visible-to-some when committed, and eventually becomes visible-to-all as other concurrent transactions finish. Some systems might in fact keep around some obsolete versions, visible-to-none, after a newer visible-to-all version exists.

When reading, a transaction first considers its own visible-to-one versions, then those visible-to-some – considering the timestamp – and finally those visible-to-all. This process, which obtains correct versions for all data items requested by some transaction, is the *snapshot reconstruction* and is the focus of this paper.

This distinction of versions in terms of visibility is not how most multi-version systems are described but is key to our insight in Section 3. Instead, systems are usually described in terms of strategies used to physically store different versions. As an example, PostgreSQL keeps them all in the main heap/file, explicitly tagged with t_xmin and t_xmax that can be compared to current visibility boundaries, termed the *snapshot*. This avoids copying old data when new versions are added, at the expense of keeping obsolete versions until vacuumed [27, 29]. Oracle labels versions with the *system change number* (SCN) [8, 11] and these reside: in the main heap/file, while visible-to-one and locked, latest visible-to-some, or if visible-to-all; other visible-to-some versions are kept separately in *rollback segments*. This optimizes for short-lived transactions, where a new version quickly becomes visible-to-all. A different example is provided by Spanner, which keeps visible-to-one versions directly in the client in unlogged structures and takes advantage of versioning in BigTable to manage committed versions, visible-to-some or visible-to-all [14].

2.3 Simplifying assumptions

Besides snapshot reconstruction, Snapshot Isolation requires precluding concurrent updates to the same item. As an example, Oracle and PostgreSQL rely on aborting transactions in lock queues on commit to ensure that the first committer wins. In distributed systems, such as Omid [19], this is achieved with a centralized validation server. A recovery mechanism is also required and usually relies on logging to ensure atomicity and durability. In this paper, we omit both of these important issues and focus exclusively on the read path for snapshot reconstruction.

We make the additional simplifying assumption of not considering the ability of a transaction to read its own writes, i.e., we ignore visible-to-one (or temporary) versions during snapshot reconstruction. Moreover, we assume that all writes are done atomically at commit time, as this simplifies representation and the manipulation of timestamps. Our proposal could be extended to accommodate such possibilities, although the current simpler form would already be useful and is actually how some systems work [14].

a) S

k	*v*
k1	v1
k2	v2
k3	v3
k4	v4

b) S_Cache

k	*v*	*from*	*to*	*deleted*
k1	v01	1	4	false
k1	v10	5	20	false
k1	v100	21	∞	false
k3	$\perp$	4	∞	true

c) S_Snapshot

k	*v*
k1	v10
k2	v2
k4	v4

Fig. 1: Example of the conversion of structure S and resulting snapshot for transaction T (T's starting timestamp = 15).

3 Proof-of-Concept

3.1 Version representation

The first pillar of our proposal is the use of regular tables or collections to hold versions with different visibilities according to Section 2.2, in contrast to using custom data structures encapsulated within a transactional storage manager layer. In detail, we separate visible-to-all (or storage) from visible-to-some (or cache) versions. The approach could be extended by considering a third table or collection for visible-to-one (or temporary) versions, which we are not addressing in this proof-of-concept.

Our key insight, which makes our proposal suitable for a polystore and compatible with a wide spectrum of datastores, is the following: *It is not necessary to keep individual version numbers for visible-to-all (or storage) versions.* The reason for this is that, by definition, all these versions are visible to all transactions unless overwritten. Therefore, their final visibility depends only on whether reconstruction picks up a more recent version while traversing cached (visible-to-some) versions. In other words, it is as if we consider that all storage (visible-to-all) versions are implicitly labeled with $ts - 1$, where ts is the oldest version in cache (visible-to-some).

The first corollary is that a transactional update and query system can be layered on top of an existing datastore without changing its content, in particular, without polluting data with additional version meta-data or multiple versions for each item, which would break compatibility with existing non-transactional applications. The second corollary is that the datastore does not need to be able to filter versions, which is hard or even impossible to do in pure key-value stores. In fact, previous transaction isolation systems that can be layered on existing datastores, such as Spanner or Omid, assume that the datastore can hold and filter versions or, in the latter, store additional version meta-data with each item.

In detail, our general approach is that for each storage table (S) in any of the supported datastores, we create an additional table for the corresponding visible-to-some versions (S_Cache). The cache accommodates data with the original schema plus three extra fields: $from$ and to, which specify a record's validity, and *deleted*, which identifies deleted records. The primary key for this table is composite, with the original key in the base storage table and $from$. As this table is not used by non-transactional applications, and only indirectly by

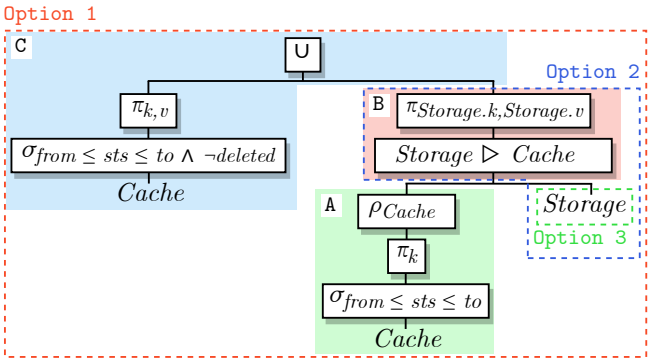

Fig. 2: Logical plan for snapshot reconstruction.

transactional applications, the additional data do not create a compatibility issue.

Figure 1 provides an example. Figure 1(a) shows some base storage table S with key k and value v. Depending on the application and the underlying datastore, both k and v can be composite values. The base table contains items with keys $k1$ to $k4$ with corresponding base values $v1$ to $v4$. Figure 1(b) shows the version cache table, added by our proposal. In detail, S_Cache shows that the value for $k1$ has been updated three times: $v01$ is valid from timestamp 1 to 4; $v10$ from 5 to 20; and $v100$ from 21. We can also see that $k3$ has been removed by version 4.

3.2 Snapshot reconstruction

The second pillar of our proposal is that we describe snapshot reconstruction for each transaction as a query to the common query engine. This is made possible by representing versions of items with different visibility as regular tables or collections.

Figure 2 outlines the logical query used to reconstruct each table in a transaction's snapshot. One branch (A) finds out which keys in the cache (visible-to-some) are relevant considering the current transaction's starting timestamp sts. These keys are used to filter out the corresponding rows from storage (B). The result is merged with values that were updated or inserted, from the cache (C). A complex query involving multiple tables requires executing this logical plan for each table.

Figure 1(c) shows the example of the reconstructed snapshot for a transaction reading from starting timestamp $sts = 15$. Records selected in each table are highlighted in green, and tombstones hiding items in red. In detail, $k1$ has been recently updated and the appropriate value corresponding to the starting timestamp of 15 is selected from S_Cache, avoiding an even more recent value with timestamp 21. $k3$ is present in S_Cache as a tombstone and thus is removed from the snapshot. Finally, $k4$ and $k2$ are obtained from the base storage table.

In short, by using a query for reconstructing the snapshot, we are able to provide isolation while, at the same time, provide a simpler alternative to specialized transactional layers or modifications to multiple datastores. It is, however, interesting to determine to what extent the resulting performance is acceptable.

3.3 Execution alternatives and optimization

The attainable performance is related to the possibility of finding an optimal physical plan for the proposed reconstruction query. Defining snapshot reconstruction as a query at the common query engine level opens up the possibility of alternative physical plans, leading to decisions by the database administrator and the automatic optimizer.

The key decision is the placement of the version cache table relative to the original storage table. Ideally, it would be placed right next to the original storage table, i.e. the same datastore, providing optimal data locality and enabling the entire reconstruction plan to be pushed down. However, since the underlying query engine might not support joining the different structures, this solution is not viable in all cases. Therefore, the version cache can be placed in a different datastore, that should be selected to provide optimal performance for the required operations. In systems such as CloudMdsQl, auxiliary tables can be stored in the common query engine itself, instead of an external datastore.

The next decision is how the logical query plan is distributed across the common query engine and external datastores. Depending on where each cache table is placed relative to the corresponding storage table and the capabilities of the query engine in the external datastore, there are three main options for what can be delegated to the datastore, depicted by dashed lines in Figure 2: (1) the entire plan, (2) filtering out irrelevant keys, (3) or nothing.

Finally, when considering snapshot reconstruction sub-plan as part of a larger query, an optimizer should be able to globally reorder and select physical operators. For instance, when executing a join operation, the query engine might opt for first joining the caches for different tables and obtain an empty result, thus avoiding the need to filter the storage. To quickly assess if these alternatives have a substantial impact on execution time, which justifies using an optimizer, and if the resulting overhead is tolerable, we resort to an experiment.

4 Experiment

We use a polystore inspired by CloudMdsQL [6], with MongoDB and Cassandra as datastores. Briefly, queries are written with a low code visual builder or the corresponding SQL-like language, with embedded native queries for different datastores. The common query engine is based on PostgreSQL, using the FDW interface for datastore wrappers. This system includes custom wrappers for Cassandra and MongoDB that optimize filter push-down, by combining them with native query languages. While MongoDB's aggregation pipeline is expressive enough to build the entire snapshot natively, the same is not possible in

Table 1: Performance of the different plans with MongoDB and Cassandra.

Query type	**MongoDB**								**Cassandra**				
	Baseline	Overhead (%)							Baseline	Overhead (%)			
	(ms)	*NG*	*NL*	*FJ*	*LJ*	*NI*	*NA*	*OD*	(ms)	*FJ*	*LJ*	*NI*	*OD*
Select all	11682	17	151	5	7	6	8	5	15702	2	6	1	1
Filter	13	9	10	14	11	12	46	12	15	1	1	1	0
Small join	13	5	8	100	94	98	95	94	28	4	9	10	8
Large join	15266	15	156	6	10	7	5	6	19740	2	5	1	2
Aggregation	299	171	5.7K	1.4K	1.4K	1.4K	93	1.9k	8842	7	14	7	10

Cassandra, and as such it relies exclusively on the common query engine to join the cache with the storage.

Therefore, we have multiple steps where the query is transformed and possibly optimized: (1) in the initial translation to PostgreSQL SQL; (2) within PostgreSQL itself; (3) in the wrapper; and finally (4) in the datastore itself. We use step 1 to determine placement and step 3 to push-down selections and projections. We have, however, limited control of step 2 in how we re-write the query in step 1 or how we provide statistics back in step 3. We deployed the system on two Google Cloud instances (8 N1 vCPUs, 8GB RAM, 500GB SDD), located on `us-east1` and `us-east4` (RTT of 11ms), one hosting the query engine and the other the MongoDB or Cassandra datastore.

Our experiment consists in running various simple queries – *select all* (returns all rows), *filter* (returns one row), small join (joins one row), large join (joins all rows), and *aggregation* (performs a sum) – on TPC-C's *order_line* and *item* tables, stored in both MongoDB and Cassandra. By manipulating placement of tables and the common query engine, we obtain several physical plans. When using MongoDB these are: native Group (*NG*, equivalent to Postgres's Order+Distinct [2]); and native Lookup (*NL*, equivalent to Postgres's Left Join [3]). Plans 2 and 3 make use of Full Join (*FJ*), Left Join (*LJ*), Not In (*NI*), Not Any (*NA*), and Order+Distinct (*OD*). For Cassandra, given the absence of a native query engine, we use only FJ, LJ, NI, and OD.

Table 1 displays the read overhead comparatively to the transaction-less alternative. The first conclusion is that different physical plans have a profound impact on query execution time, up to 58× worse in one scenario! Most strikingly, *it is clear that different plans are optimal in different scenarios*, which is a compelling argument for the use of an optimizer.

Finally, these results let us infer a transactional read overhead under 10% for most cases with both datastores, which compares favorably to the measured cost of corresponding transactional mechanisms in a traditional SQL database system [20]. The exception is the aggregation query with MongoDB. While the Not Any approach can execute the partial aggregation natively in MongoDB, greatly reducing data transfer, the engine filters the storage with the cache keys using, for this particular case, a suboptimal index scan implemented with the

keys' bounds. Since each of the thousands of keys are completely different, the scan will consider thousands of bounds. For this case, a better alternative would be a *hash anti join*, which should bring the overhead closer to the other queries.

5 Discussion

In this paper we address a challenge, transaction isolation in polystores, that has seen very little previous attention, even if identified as a key research challenge [28, 30]. Transactional support is a challenge even in multi-model datastores, naturally more integrated than polystores, where support for multi-model transactions seems to be non-existent [24].

The main competing approach for transactional isolation in polystores is Polypheny-DB [32]. In contrast to our proposal, which aims at running read-only transactions with little interference and at fine-grained conflict resolution for update transactions, Polypheny-DB assumes two-phase locking with coarse granularity, which limits concurrent updates and makes them conflict with read-only transactions.

We are also aware of a different proposal that has been prototyped in CloudMdsQL [22], as part of the same research project. Like our current proposal, it aimed at Snapshot Isolation and fine-grained conflict resolution but it relied on the implementation, from scratch, of a custom wrapper, or even core changes, for each datastore. It also assumed the version cache is always co-located, which often resulted in changing of the native schema.

Similar motivation can be found in DeltaLake [7], aimed at incremental loading or correction of data in a data lake with coarse-granularity. In contrast to our proposal, it is not aimed at polystore but only at data in Parquet files directly managed by Spark. Therefore, snapshot reconstruction in DeltaLake boils down to reading the right subset of file fragments, making updates and removals very costly as a new version of affected files needs to be written. It is also not clear how it would be extended to polystores.

Our approach is to define transactional isolation in terms of additional tables, managed themselves within the polystore, and generic queries that can be mapped to a common query engine layer and multiple datastores. This takes advantage of query optimization to achieve the optimal execution plan for each particular polystore configuration. In fact, a preliminary experiment shows that the overhead of transactional isolation is comparable to what has been measured in traditional SQL systems [20].

An interesting outcome of this experiment regards the feasibility of the proposed approach: To what extent keeping updated versions in a separate table can be reconciled with full use of the interface of each datastore? Namely, can a native query for some datastore always be modified to account for such versions? In our experiment, this is very easy to do with a key-value store such as Cassandra, where returned values can easily be replaced. Our experience with MongoDB is different: We cannot easily patch the result from a native query, which can be an arbitrarily complex "aggregation pipeline." On the other hand,

this makes MongoDB expressive enough that the query can be modified to the reconstruction by itself. We postulate that this might be generally true: Whenever the native query engine is complex enough to make patching the result hard, it should be expressive enough to be itself used for reconstruction.

The main threat to the validity of our experiment is that we omit the write path. We expect to approach this by defining how updates on a view should be translated to changes in the cache table, which can be implemented, for example, using INSTEAD OF triggers or rules [4]. This possibility is limited by known bounds on updatable views as the reverse mapping may not always exist [12] and the challenges of translating an update u to a view V into a set of updates U to the underlying data D, namely [15, 16]. Additionally, we have to consider multiple data models and, in the CloudMdsQL, the effect of ad hoc views, for which we can resort to bi-directional transformations, with weaker guarantees on update properties [10, 25]. Finally, we have to coordinate the recovery of heterogeneous multi-statement transactions with the various recovery guarantees of individual datastores.

We can thus identify several lessons learned and outstanding challenges for transactional polystores:

Optimization and DBA are needed. We have shown that structuring snapshot reconstruction as a data processing problem allows optimization (different plans are optimal in different conditions) and provides an opportunity for a DBA to intervene.

Useful for different datastores. Datastores with a more complex QE make it harder to store changes and reconstruct the snapshot outside (e.g. MongoDB) than simple key-value stores, but on the other hand, they make it easier to use their own QE for reconstruction, which makes the approach feasible across a large spectrum of datastores.

Datastore interfaces matter. It is highly relevant that the data-store language is amenable to processing and manipulation, without having to rewrite a lot. For instance, MongoDB's aggregation pipeline is ideal and much more complex than the SQL-like in Cassandra. It is thus a challenge to provide this and still be user-friendly for writing native queries.

Various isolation criteria are possible. Polystores are inherently distributed and likely over a WAN, making strict snapshot isolation problematic. Moreover, it is likely that one size does not fit all applications is also true in terms of isolation level. A possible alternative is TOPSI [18].

Update processing is an open problem. Updates issued at the common QE level are issued on views. This means that they have to be translated back to the original data model for the underlying datastore.

Interaction with native readers and writers is an open problem. Our proposal provides transactional isolation when all readers and writers access datastores through the common query engine. A consistent view of a prefix of updates to native readers should also be possible by judiciously scheduling checkpointing operations. It is unclear, however, if it is possible to do the reverse: Allowing native clients to update datastores without disturbing isolation.

References

1. Distributed transaction processing: The xa specification (1991), https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf
2. Mongodb 4.4 manual - aggregation pileline stages: $group (2020), https://docs.mongodb.com/manual/reference/operator/aggregation/group/
3. Mongodb 4.4 manual - aggregation pileline stages: $group (2020), https://docs.mongodb.com/manual/reference/operator/aggregation/group/
4. Postgresql documentation - 40.4. rules on insert, update, and delete (2020), https://www.postgresql.org/docs/13/rules-update.html
5. Transaction management - the neo4j java developer reference v4.3 (2020), https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf
6. Alonso, A., Abreu, J., Nunes, D., Vieira, A., Santos, L., Soares, T., Pereira, J.: Building a polyglot data access layer for a low-code application development platform - (experience report). Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-50323-9_6, https://doi.org/10.1007/978-3-030-50323-9_6
7. Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., undefineduszczak, A., undefinedwitakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., Xin, R., Zaharia, M.: Delta lake: High-performance acid table storage over cloud object stores. Proc. VLDB Endow. **13**(12), 3411–3424 (Aug 2020). https://doi.org/10.14778/3415478.3415560, https://doi.org/10.14778/3415478.3415560
8. Bamford, R.J., Jacobs, K.R.: Method and apparatus for providing isolation levels in a database system (Feb 9 1999), uS Patent 5,870,758
9. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. ACM SIGMOD Record **24**(2), 1–10 (1995)
10. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 338–347 (2006)
11. Burleson, D.K.: Oracle internals: tips, tricks, and techniques for DBAs. CRC Press (2017)
12. Codd, E.F.: Recent investigations into relational data base systems. Tech. Rep. RJ1385, IBM (4 1974)
13. Cooper, J., James, A.: Challenges for database management in the internet of things. IETE Technical Review **26**(5), 320–329 (2009)
14. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS) **31**(3), 1–22 (2013)
15. Dayal, U., Bernstein, P.A.: On the updatability of relational views. In: VLDB. vol. 78, pp. 368–377. Citeseer (1978)
16. Dayal, U., Bernstein, P.A.: On the correct translation of update operations on relational views. ACM Transactions on Database Systems (TODS) **7**(3), 381–416 (1982)
17. Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.: The bigdawg polystore system. SIGMOD Rec. **44**(2), 11–16 (Aug 2015). https://doi.org/10.1145/2814710.2814713, https://doi.org/10.1145/2814710.2814713

18. Faria, N., Pereira, J.: Totally-ordered prefix parallel snapshot isolation. In: Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data. PaPoC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3447865.3457966, https://doi.org/10.1145/3447865.3457966
19. Gómez Ferro, D., Junqueira, F., Kelly, I., Reed, B., Yabandeh, M.: Omid: Lock-free transactional support for distributed data stores. In: 2014 IEEE 30th International Conference on Data Engineering. pp. 676–687 (2014). https://doi.org/10.1109/ICDE.2014.6816691
20. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: Oltp through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. p. 981–992. SIGMOD '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1376616.1376713, https://doi.org/10.1145/1376616.1376713
21. Hellerstein, J.M., Stonebraker, M., Hamilton, J.: Architecture of a database system. Found. Trends Databases **1**(2), 141–259 (Feb 2007). https://doi.org/10.1561/1900000002, https://doi.org/10.1561/1900000002
22. Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., Pereira, J.: CloudMdsQL: querying heterogeneous cloud data stores with a common language. Springer Distributed and Parallel Databases pp. 1–41 (2016). https://doi.org/10.1007/s10619-015-7185-y
23. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
24. Lu, J., Holubová, I.: Multi-model databases: a new journey to handle the variety of data. ACM Computing Surveys (CSUR) **52**(3), 1–38 (2019)
25. Macedo, N., Pacheco, H., Cunha, A., Oliveira, J.N.: Composing least-change lenses. Electronic Communications of the EASST **57** (2013)
26. Schultz, W., Avitabile, T., Cabral, A.: Tunable consistency in mongodb. Proc. VLDB Endow. **12**(12), 2071–2081 (Aug 2019). https://doi.org/10.14778/3352063.3352125, https://doi.org/10.14778/3352063.3352125
27. Stonebraker, M.: The design of the postgres storage system. In: Proceedings of the 13th International Conference on Very Large Data Bases. p. 289–300. VLDB '87, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1987)
28. Stonebraker, M.: The case for polystores. ACM SIGMOD Blog (2015), https://wp.sigmod.org/?p=1629
29. Suzuki, H.: The internals of postgresql: Chapter 5 concurrency control (2021), https://www.interdb.jp/pg/pgsql05.html
30. Tan, R., Chirkova, R., Gadepally, V., Mattson, T.G.: Enabling query processing across heterogeneous data models: A survey. In: 2017 IEEE International Conference on Big Data (Big Data). pp. 3211–3220 (2017). https://doi.org/10.1109/BigData.2017.8258302
31. Vogels, W.: Eventually consistent. Communications of the ACM **52**(1), 40–44 (2009)
32. Vogt, M., Hansen, N., Schönholz, J., Lengweiler, D., Geissmann, I., Philipp, S., Stiemer, A., Schuldt, H.: Polypheny-db: Towards bridging the gap between polystores and htap systems. In: Gadepally, V., Mattson, T., Stonebraker, M., Kraska, T., Wang, F., Luo, G., Kong, J., Dubovitskaya, A. (eds.) Heterogeneous Data Management, Polystores, and Analytics for Healthcare. pp. 25–36. Springer International Publishing, Cham (2021)