Sharding – Some Dirty Little Secrets

Large data sets or high throughput applications challenge any single-node database. Large capacity exceeds the server's storage. Heavy workloads exhaust CPU capacity, I/O resources, or memory (RAM). Upgrading servers with more storage, memory, and faster CPU might solve this challenge. However, this is expensive, and there is a limit to the capacity or workload that a single server can handle.

The more contemporary approach to solve this issue is to horizontally scale the infrastructure out – i.e. get a lot more power than a single large server can provide by combining the capacity, I/O power of many disk devices, and compute power of multiple servers in a cluster. A common form of scale-out infrastructure for databases is scale-out sharding. With scale-out sharding, data is divided across multiple servers ("shards") in a cluster according to a sharding criterion. Each server operates as a small independent database that can easily handle its part of the total data. Scale-out sharding requires the application code to be well-aware of the exact data placement across the local independent database servers (shards) such that it will be able to manage the workload across those shards.

Contrary to what is sometimes popular belief, scale-out sharding configurations typically do not offer the notion of a large, single, database. Scale-out sharding does not try to align many nodes to act like one big database. Rather, each node operates as an independent sub-database.

As a result, some basic database functionality that can be expected from a single node database cannot be achieved in multi-node sharded databases. For example, cross-shard-boundaries transactions and analytics operations may not work or not work well. In addition, sharded databases may suffer other issues including, for example, per-node performance bottlenecks, inefficient workload execution, and even volume-skews surpassing nodes' available capacity. Changing sharded database cluster configurations, like adding or removing servers, is generally challenging -- not least because of the excessive amounts of data movement across the cluster nodes that this might kick off, the related disruptions to operations, or the significant modifications that developers inherently must make in the application code.

Sharded databases cannot guarantee distributed ACID

ACID properties are intended to guarantee that database transactions are valid. If data accuracy of database operations is important, then the database should guarantee ACID. The absence of ACID means that there are no guarantees of atomicity, consistency, isolation, or durability.

Many a seasoned developer might now nod and declare that it surely is possible to build applications on modern databases that do not guarantee ACID. This requires writing code and logic that deals by itself with the lack of ACID, and somehow guarantees correctness and ordering of operations at the application level. Of course, this is not necessarily possible, and inevitably introduces tradeoffs and complexity in the process and the application logic. Attempts to maintain ACID-like guarantees at the application layer are known to be hard or even impossible, and in any case complex, painful, and usually buggy.

What does this mean for multi-node databases and ACID? When a single-node database that guarantees ACID is expanded to multiple servers (for example by scale-out sharding), it is expected that the multi-node database will also provide the same ACID guarantees, i.e. across all the shards in the entire cluster. However, this is generally not the case.

Let's have a look at some examples that illustrate this.

A DISTRIBUTED TRANSACTION WITH ONE SUCCESSFUL AND ONE UNSUCCESSFUL LOCAL COMMIT

In our first example the application runs on a sharded database. The application executes a transactive operation that involves two shards, Shard-1 and Shard-2. Since shards are independent databases, the application needs to take responsibility for executing this operation as a distributed transaction. To realize this, the application needs to execute the distributed transaction as two local-shard transactions, one on each shard[1]. Each shard executes its own local transaction and does not have any way to realize that its local transaction is in fact part of a distributed transaction. It is therefore the application's responsibility to manage the execution of the distributed transaction.

Suppose the application progressed with the execution of the distributed transaction and is now ready to commit. To commit the distributed transaction, the application instructs two independent commits, each on a different shard. Unfortunately, it is impossible to guarantee that both local commits will succeed[2]. As a result, one local transaction may successfully commit while the other local transaction fails. The result is data inconsistency. To illustrate this, see the figures showing a situation in which the local transaction on Shard-1 succeeds to commit, but the local transaction on Shard-2 fails to commit.

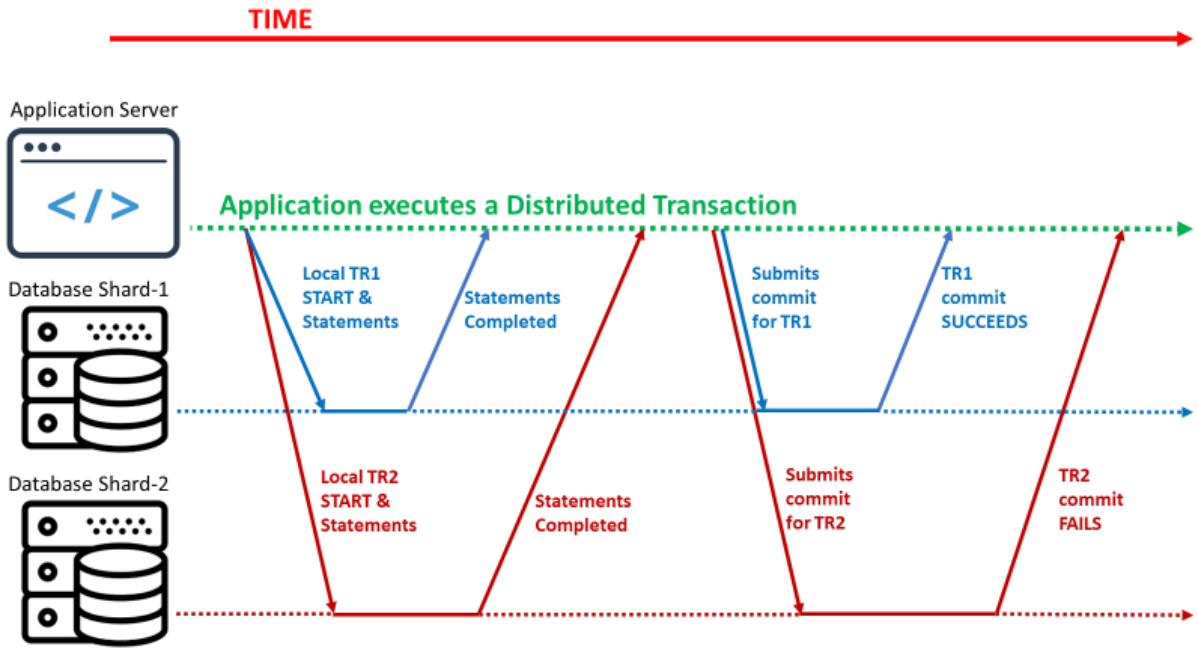

Figure 1 Application executes distributed application as two local transactions

The distributed transaction was only able to commit "partially". Local transaction TR1 on Shard-1 commits successfully, while local transaction TR2 on Shard-2 fails to commit.

[1] Each local transaction consists of one or more SQL statements.

[2] There is a variety of reasons for a local-transaction commit to fail. For example, conflicts with other local-transactions executing on that shard (like first-committer-wins check, deadlock in which the database elected to purge our local-transaction, etc.), resource issues, network outage, or hardware failure could all prevent successful commitment of the local-transaction.

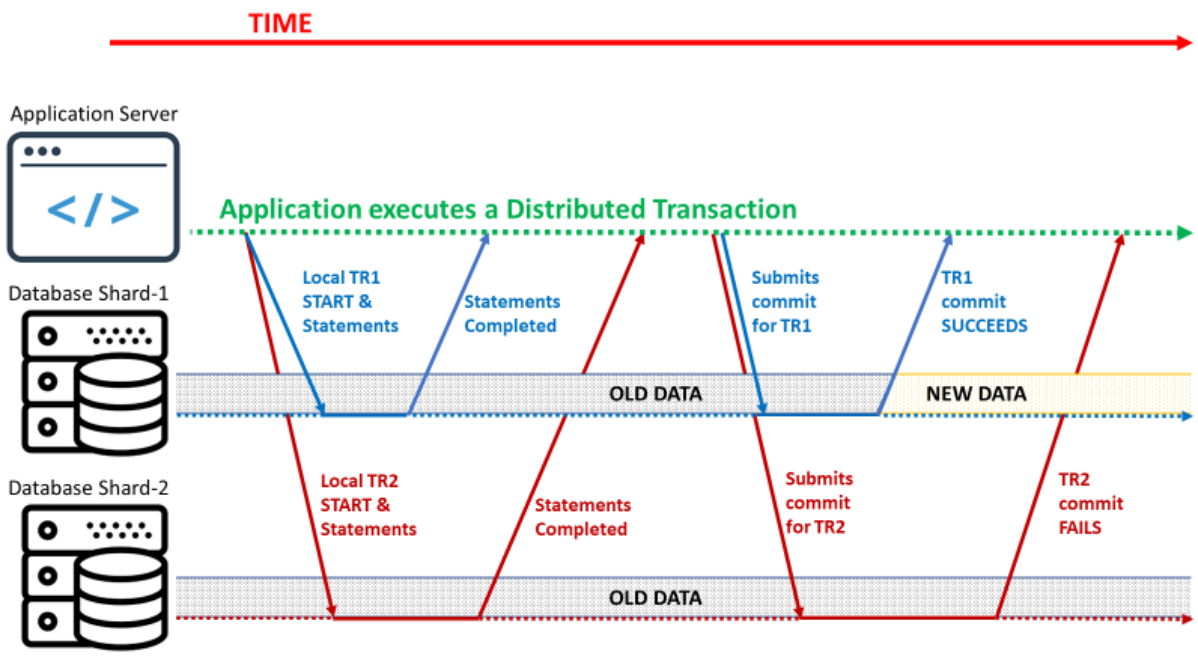

Figure 2 New data appears only on Shard-1

The new data appears only on Shard-1 while the old data on Shard-2 remained unchanged. This created a “window of inconsistency” during which committed, new, data on Shard-1 and old, unchanged, data on Shard-2 were visible at the same time.

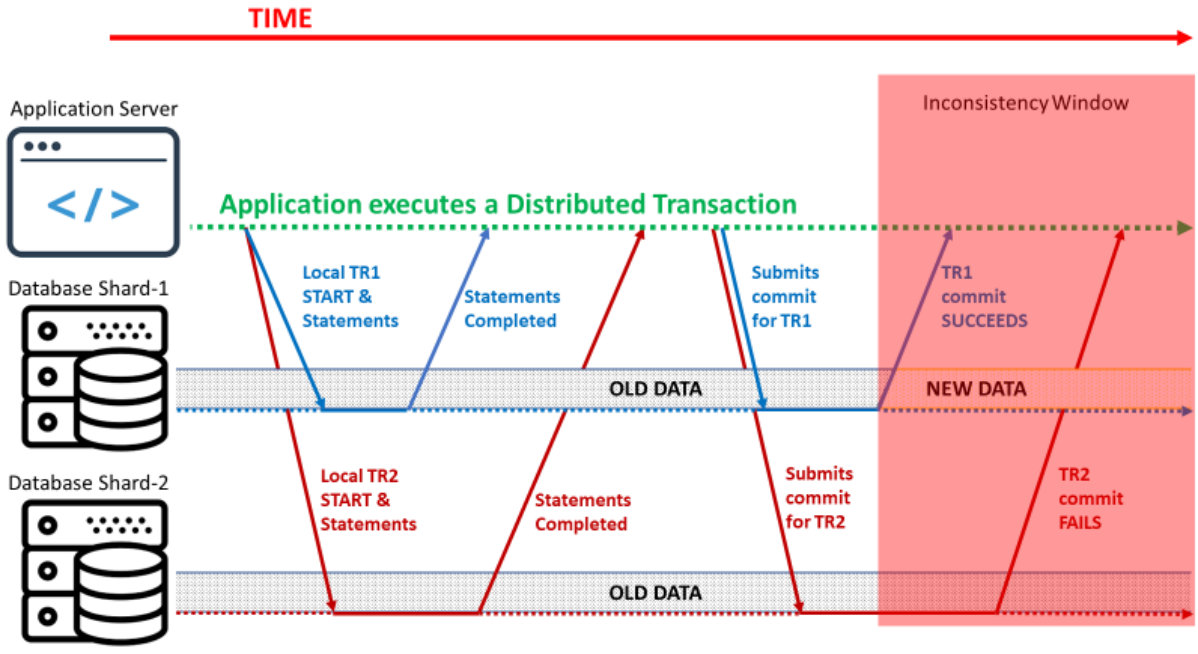

Figure 3 Combination of new data on Shard-1 and old data on Shard-2 creates a window of inconsistency

As long as the application cannot remedy this inconsistent state, this window of inconsistency remains open[3]. This is a problem since it violates the distributed transaction's atomicity. Atomicity[4] is a fundamental ACID property since its violation could lead to various data corruptions and inconsistencies.

The partial commit of the distributed transaction caused a window of inconsistency that remains open until the application can resolve the scenario that caused it. How can the application solve this situation and close the window of inconsistency? One way would be to overwrite the new data, that was committed by the local transaction on Shard-1, with the old data – this is called a "roll-back". Alternatively, the application can attempt to retry the failed local transaction on Shard-2, thus writing the new data where the local transaction on Shard-2 previously failed to commit – this is a "roll-forward". Each of these approaches has a good chance of succeeding and thus to close the window of inconsistency. However, there are cases where none of the strategies can succeed, which causes the inconsistent state to remain stuck forever.

Roll-forward

The database could decide to roll-forward (retry) the unsuccessful transaction on Shard-2. In this case, the local transaction on Shard-2 could easily fail to commit due to conflict with another transaction that might have executed in the meantime on Shard-2 and have modified the database in a way that will not allow the original local transaction to commit, even when re-executed…

Roll-back

Databases can only roll-back local transactions that have not committed yet. It is not possible for the database to roll-back an already committed local transaction. Hypothetically, the application itself could, somehow, force the database to perform a roll-back after a commit – but this is hypothetical since this functionality violates ACID and is thus generally not supported by databases. Alternatively, the application could perform a roll-back-*like* operation by remembering the old data of Shard-1 and then submitting a new local transaction that will overwrite the recently committed "new" data with the old data.

Interestingly, like in the roll-forward scenario, there may be cases where it is categorically not possible to commit the roll-back-like local transaction at Shard-1. For example, if conflicting transactions were executed in the meantime.

[3] During the window of inconsistency, unrelated other transactions can read potentially wrong data – and will rely on that wrong data.
[4] With atomicity, multiple operations of the same transaction must either succeed or fail together.

DISTRIBUTED TRANSACTION WITH TWO SUCCESSFUL LOCAL COMMITS

Consistency issues can occur even if both local transactions commit successfully. For our next example, let's look at such a situation where both local transactions, on both local shards, commit successfully.

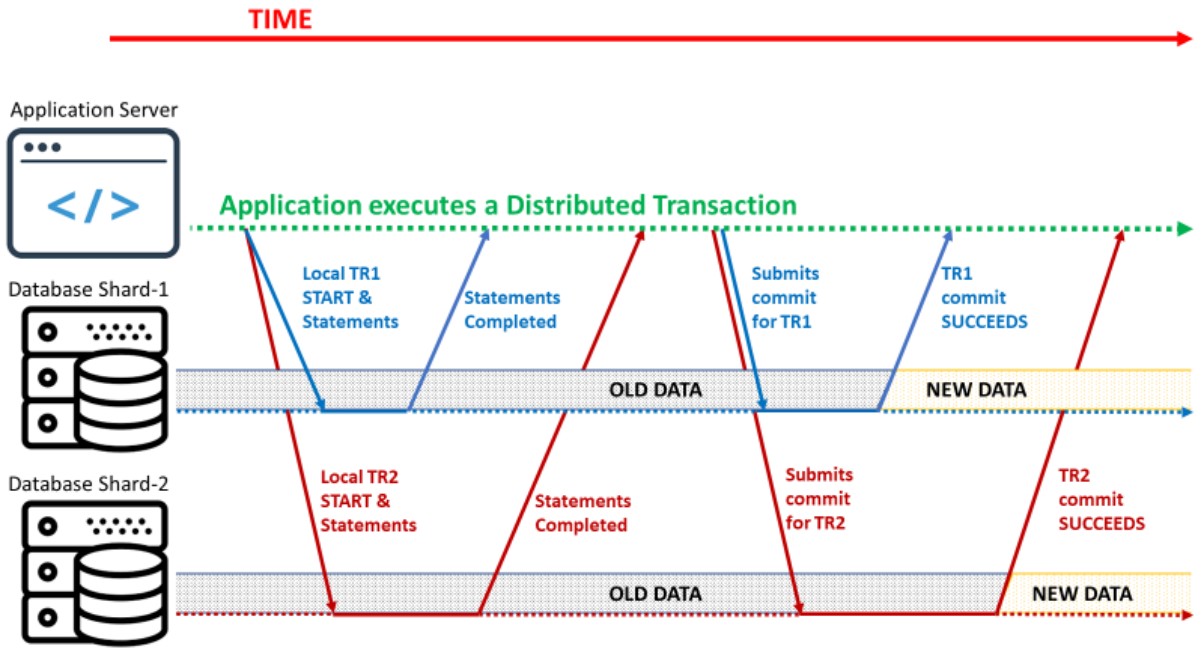

Figure 4 Both local transactions commit successfully on both shards

Since both local transactions committed successfully, the application's distributed transaction committed successfully. Unfortunately, we still cannot guarantee ACID. Why? Both local transactions executed independently of each other. There was no way to guarantee that both committed at the same time exactly. Hence the data written by the first local transaction[5] could have been visible after the first local transaction successfully committed and before the second local transaction committed.

[5] Either shard may be the first to commit its transaction.

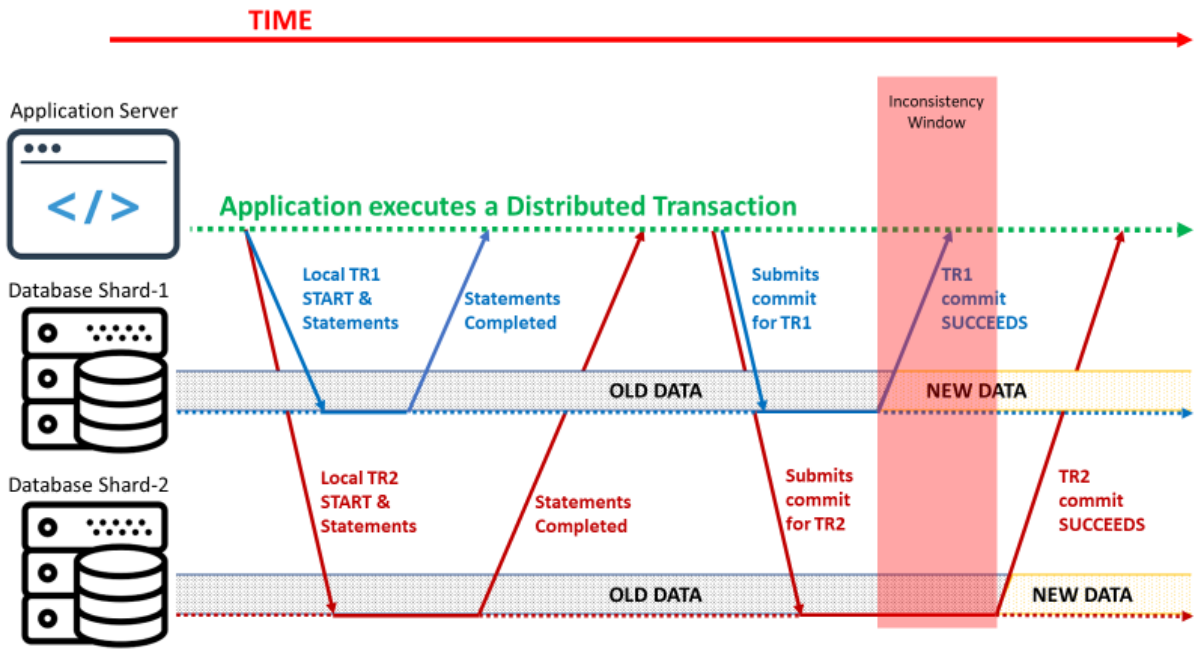

Figure 5 Both local transactions commit successfully, but not at exactly the same time, creating a Window of Inconsistency

This exposed other transactions to a mixture of new (committed by the first local transaction) and old (yet uncommitted by the second transaction) data. Hence even two successful local transactions can break the atomicity[6]. No atomicity means no ACID.

In addition, the visibility of the mixture of new and old data might cause the "reader's" application code to not only read inconsistent (non-atomic) data, but also to cut the wrong decisions based on such partial data. This can in turn lead to further data inconsistencies in the database, lead to bugs in the reader's code, and also lead to crashing of the application.

Intermediary Summary: Sharded databases cannot guarantee distributed ACID

So far, we have shown the importance of transactions and guaranteeing ACID. Guaranteeing ACID is imperative to ensure the validity of applications' transactions, and to guarantee data consistency.

When expanding a single-node database that guarantees ACID to multiple independent database nodes by scale-out sharding, it is expected that the multi-node database will maintain those same ACID guarantees across the entire cluster. Having local ACID guarantees at the level of individual shards is not sufficient to also guarantee ACID at the distributed, multi-node database level. As illustrated in the examples above, expanding a single-node database to a cluster of database nodes by scale-out sharding breaks the ability to maintain the same ACID guarantees in the cluster. As the number of nodes in a cluster grows, both these and other types of ACID-related challenges become increasingly more complex and problematic. This has a direct impact on the application developer.

[6] Note: Interestingly, the fact the two transactions have not committed at exactly the same time is just one problem out of two – each of which on its own is sufficient for atomicity violation. The second fact has to do with how ACID databases generally guarantee consistent reads – i.e. the ability of a later transaction to read a consistent set of records. Without getting into all the details, databases typically provide that fundamental capability by either taking relevant internal locks or by using snapshots (or a combination of the two). In our case, both techniques would have needed to somehow work in perfect coordination across the two shards. In other words, to achieve consistent reads across the two shards we would need distributed lock management and/or distributed snapshots to be built into the database. Since each of the shards is essentially an independent single-node database, these techniques are not available…

When developers cannot rely on the underlying data store they have no choice and must assume responsibility for the data infrastructure[7], in addition to their ownership of the application logic.

This inability to rely on the underlying database has additional development, operational, and infrastructure cost implications. Larger development teams are required, with more experienced, more expensive development engineers that have deeper system knowledge. The design process is more complex since a variety of corner cases must be taken into consideration. Code is more complex, and a lot of time must be spent on activities like tracking down anomalous transactions, quality assurance, and testing. Visible but unnoticed costs of suboptimal design at the database level might result in additional cost implications to the business since unexpected application behavior might force it to spend more money on refunds, investigations, customer satisfaction, or even compensations.

If you are building an application, you do not want to be in the business of designing and developing concurrency control and durability yourself. Even experienced developers agree that it is much easier to rely on the database than to attempt to achieve ACID at the application level. Having to implement ACID and other data integrity guarantees in the application code will slurp up a significant amount of development and financial resources. As a result, the application code will be a lot more complex, often resulting in more bugs and unpredictable development schedules. Being able to rely on the underlying database for ACID and other data integrity guarantees results in more reliable and faster application's time-to-market.

In the next chapter we will look at another set of challenges. We will cover another class of functionality, namely query execution in sharded databases.

[7] For example, Oracle states on Oracle.com (March 2, 2017) that "[…] applications must be explicitly designed for a sharded database architecture in order to realize the benefits of scalability and availability."

Sharded databases are not well-suited for executing queries

Executing a distributed query

Let's have a look at a sharded 100-nodes database of a stock exchange. The database contains one table called Trades that is randomly distributed across the cluster's nodes (e.g., according to a hashing criterion):

- Trades: (id, share_id, seller_id, buyer_id, cost, trade_date)
  - *About 25 million new entries per day*

The database has an index on cost for the Trades table. Since each shard operates as an independent database, each shard has its own cost index, where each such index contains information only for the Trades records contained by that specific shard.

The application's query is: ***Find the details of the 1,000 trades that have the lowest total cost above $500.***

How would the application execute this distributed query, given the fact that the underlying sharded database is effectively a set of 100 independent databases, and given the fact the answer to this query does not reside in one specific node?

The most straight-forward approach would probably be:

- The application submits 100 query-transactions to each of the 100 shards, requesting each shard separately to query the 1,000 trades with the trades that have the lowest cost above 500$.
- Each of the shards executes that query by taking the following actions:
  - Access its local cost index and find the 1,000 trades with the lowest deal size that is above $500 *(typically, such an index operation is very cheap and could be achieved with a very small number of I/Os)*
  - Read those 1,000 rows from disk *(this is more expensive, costing 1,000 or more disk I/Os)*
  - Return the requested details of these 1,000 Trades-rows to the application
- At completion of the 100 query-transactions, 100,000 Trades-rows were read and transferred to the application, which keeps them in RAM
- The application now must sort all the 100,000 rows by their cost and choose the 1,000 trades with the lowest values

While it may work, it may be obvious that this is expensive, inefficient, and complex.

For comparison: If the above would have fitted in a single node database, then the execution of this query would have been a simple SQL query like this:

```
SELECT LIMIT  1000 trade_id, cost, share_id, seller_id, buyer_id, trade_date
FROM          Trades
WHERE         cost > 500
ORDER BY      cost ASC
```

In that case, the entire query would have entailed only:

- 1 transaction
- An index access for locating the 1,000 rows (very few I/Os)
- 1,000 row reads (say 1,000 I/Os)
- Transfer of 1,000 rows to the application
- No need for sort by the application

It would have been 100x cheaper in multiple dimensions, i.e. 100x less I/Os, 100x less network-traffic, 100x less transactions, 100x less RAM required by the application to store the intermediate results, etc.

Note that in this example, the number of nodes in the cluster influences the cost (i.e. resources consumed) of the query. The larger the cluster, the more expensive the execution of the operation. This is not linearly scalable and therefore very inefficient[8].

Also, the fact that the application needs to incorporate computational operations like SORT means that the application needs to execute operations that would be more naturally executed by the database.

All the above assumes that the capacity of the 100,000 rows fits in RAM. In another scenario where those rows' data would not fit RAM (e.g. significantly larger number of rows), then the application developer would have to deal with much more complicated disk-based sorting. This type of complex operations should ideally not be part of the development of an application's business logic. Databases are naturally the best suited to handle exactly these types of operations.

Executing more complex distributed queries

Now let's look at the database of a large online retailer. The database contains the following tables:

- Customers: ~1B entries
- Products: ~50M entries
- Deals: ~1B entries per day (information is kept for the last 10 years). Each deal is performed by a single user for a single product.

The database size is multiple PB. If, for example, the size of a deal record is 250 bytes, just the deals table alone is already approximately 1PB. The actual size of the database is much larger since it includes other data elements that are not of interest for our example.

The retailer would like to know the following*: **How many customers in each country bought products that cost more than 100$ and that were made in Spain, in the last 5 years?***

If the amount of data in these tables were small enough to fit a single-node database then answering this query would have been executed with a simple SQL statement like this:

```
SELECT  u.country, Count(DISTINCT u.id)
FROM    deals d
  JOIN  items i ON d.item_id = i.id
  JOIN  customers u ON d.buyer_id = u.id
WHERE   i.make_country = 'Spain'
  AND   i.price > 100
  AND   d.date > today.subtract_years(5)
GROUP BY u.country;
```

[8] There are other approaches to solve this by, for example, a distributed 100-way K-way merge. This would result in a smaller number of rows read, but dramatically increase the number of processed SQL statements and the chattiness over the network. All in all, it may end up being as expensive as the first approach discussed above. Of course this even further complicates the application logic that now needs to implement an efficient distributed K-way merge algorithm.

This example assumes that the retailer's query was provided to the database in an ad hoc manner, and that the database owner did not prepare for this type of queries in advance by, for example, by creating indexes or organizing the data in a certain way – as is sometimes possible in traditional OLAP and, especially when queries are repetitive rather than ad hoc, may help shorten time-to-data-insights.

Furthermore, the query execution of the above example, if the amount of data would have been small enough to fit in a single node relational database, would have been straightforward, too. The processing of the JOIN and DISTINCT operations in a single-node database is quite standard, though might require the database to internally execute various optimizations and other algorithms that may be sophisticated. Trying to mimic those operations and internal database algorithms, even of just a single node database, at the application level is not an easy task. That task gets even more complicated if the temporary data that is required for calculating the query does not fit in the application server's RAM.

However, as stated, the database in our example has a size of multiple PB and does not fit a single-node database. How would the application execute this query if the data is stored across 100 independent database shards?

The most naïve way would be for the application server to fetch the entire data that it needs from all the shards, and to calculate the query's results accordingly. There may be multiple strategies to perform such a calculation. It is beyond the scope of this document to detail each of these strategies, but it would be safe to say that they would require a large amount of RAM and CPU resources, and, depending on the exact strategy, significant application-server storage performance.

It is also safe to state that all strategies require the application server to fetch data of 1.8 trillion deals – representing the need to move about 400 Terabits of information over the network. This amount of network traffic would easily saturate a 10Gb ethernet card for more than ten (10) hours. This effectively renders all available strategies irrelevant, regardless of whether the required excessive amounts of RAM, CPU, and storage performance can be delivered by the application server, or not.

A less naïve approach would be for the application server to find a way to break this query calculation up into units that can be offloaded to the 100 shards. If this would be possible, it would avoid the huge data transfer that practically killed the more naïve approaches discussed above.

However, any approach to breaking the query up into 100 independent tasks, running on the 100 shards, faces near-insurmountable challenges. For example, if the application would cause each of the 100 nodes to only scan the deals data on its own shard in order to find those with items made in Spain and worth $100 or more, it would still not solve the cross-shard data movement challenge. The same node would still need to fetch all the customers information for those Spanish $100 deals that he found. Getting that customers information (most of which is not stored on that same shard) requires this shard to fetch the relevant part of the 1B customers that are spread over 100 nodes. However, since each shard is an independent database node, it does not have the capability to even know how to find this data.

Furthermore, even if it had the capability to know how to find this data, the amount of data that would have to be sent over the network would have been excessive since each user record probably will need to be fetched by each of the many nodes processing their deals. The same records would be sent many times to many nodes.

The proper way to deal with the above challenges would have been by executing a *distributed JOIN.* Unfortunately, a distributed JOIN cannot be performed by 100 independent shards that are not aware of each other and that do not have distributed processing capabilities.

Taking this example even one step further: Imagine that, in that way or another, each of the shards created a list of customers that did the Spanish products $100 deals and stored it in that same shard. Each of those 100 shards' user lists, i.e. 100 lists, now need to be subjected to a DISTINCT processing, in order to ensure that no user will be counted more than once. Unfortunately, naïve, non-distributed DISTINCT processing would also result in very large data transfers from many nodes to a single node which would make the query run for much longer.

Similarly to the JOIN part of the processing, a *distributed DISTINCT* would make the processing much faster. However, performing a distributed DISTINCT in a sharded database is as hard or impossible.

Last but not least, the above example discusses how to handle an *ad hoc* query. Even disregarding all the complexities of the above, developing code at the application level to execute ad hoc queries across a sharded database is not an easy feat by multiple other standards either.

Intermediary Summary: Sharded databases are not well-suited for executing queries

Following the discussion of the importance of transactions and ACID in the beginning of this document, we now reviewed some examples of the impact of executing queries on scale-out sharding distributed databases.

As shown, the application developer needs to do a lot of hard work and to write a lot of complex and error-prone code in the application to attempt to realize the execution of queries. Moreover, in many cases, writing code for the execution of such queries at the application level is simply not possible at all.

In the remainder of this document, we will cover a few additional challenges posed by scale-out sharding.

Some additional Dirty Little Secrets of Scale-Out Sharding

Challenges of data-based and hash-based sharding criteria

As mentioned in the beginning of this paper, with scale-out sharding, data records are divided across multiple nodes ("shards") in a cluster according to a sharding criterion. This criterion (e.g. product category) might be set by the application developer. Alternatively, the developer can decide to define the sharding-criterion based on hashing[9].

When distributing data based on a sharding criterion, related records are placed together in the same shard. For example, all products from the same category. This enables the database to leverage the fact that related data is co-located in the same shard, and to execute some logic "in shard"[10].

Hash-based sharding delivers a more balanced distribution that results in less skews, however, it would generally not be possible to benefit from the above-mentioned advantages of data-based (i.e., non-hashed) sharding criteria.

[9] Hashing is a mathematical function that takes a large number of data entities as input and distributes them in a near uniform manner across a small number of nodes. Note that it is possible to have a combination of a sharding criterion and hashing.

[10] Another benefit of co-locating related data is that those records are stored within the same disk blocks, which may improve I/O performance when multiple records are accessed together.

As there is no natural balancing between the types of data, placement based on a (non-hashed) sharding criterion might result in significant skews between shards' data volumes. Worse, the data volume of some shards might be larger than the nodes' disk capacity -- which means that either the node's hardware must be expanded or replaced, or that the scale-out cluster gets "stuck" without the ability to adequately represent all data. The only way to get out of such a "stuck" situation would then be to choose another sharding criterion altogether, and to redistribute the data across the shards accordingly[11]. A very intrusive and undesirable solution indeed.

Shards' workloads or performance can also get skewed. A shard with more rows than others may expect a larger number of transactions and/or lengthier, more expensive query processing. Even when the data placement is well-balanced, for certain workloads the application logic might have to work on one shard only, and not be able to spread the workload across more resources on multiple shards. This often causes that specific shard to become a performance bottleneck. For example, the "bicycles" shard will work hard while all other shards are idle, if data was distributed according to product category and the application's workload focuses on bicycles. It is not possible to leverage all the other servers' free CPU and RAM resources to run the workload on multiple servers in parallel rather than just on that single - bottlenecked – "bicycle" shard.

Addition and removal of servers and/or disks

Adding or removing servers or disks to/from a sharded database's cluster is often disruptive. If the data was distributed by hashing, then the server addition or removal will typically require rehashing which will cause excessive amounts of data movement of all, or almost all, the existing data across shards[12]. This is expensive in terms of network utilization and disk operations, and degrades performance.

Data distribution based on a non-hashed sharding criterion normally results in having an as-large-as-possible data volume per shard. Having more data per shard enables applications to execute more operations within a shard rather than across shard boundaries - thus limiting the exposure to the many downsides of cross-shard boundaries operations. Adding or removing servers in a sharded database with data that was distributed based on a non-hashed sharding criterion, especially when its shards have a lot of data, might require a change of the database's sharding criterion to a completely different type of sharding criterion.

Why? Take for example a database with 50 nodes, where the sharding criterion causes the data of each of the country's 50 cities to be placed on a separate node. Now expand the cluster to 80 nodes. The dataset only has 50 cities because the country in question only has 50 cities, and there simply do not exist 80 cities there. Another sharding criterion needs to be chosen (rehashing) to distribute the data across 80 nodes. Note that this is one of the reasons why developers will often prefer not to change the number of nodes in the first place.

However, when the database data grows, they may have no other choice, as there is a limit to how much one could scale-up a single node by increasing its RAM, CPU and disk resources.

Application code must be fully aware of the sharding data distribution and placement. For example, Oracle's documentation states that "OLTP applications must be explicitly designed for a sharded database architecture in order to realize the benefits of scalability and availability" [13].

---

[11] This scenario mainly assumes direct attached storage in a shared-nothing architecture. However, also in case of shared storage (SAN or NAS) skews can cause a node to work too hard due to having much more data.

[12] The described excessive amounts of data movement might in certain cases and according to certain methods be mitigated.

[13] Source: Oracle.com, March 2, 2017

Adding or removing nodes is disruptive since the application must know exactly where to send each query. As mentioned above, changes in the cluster configuration may easily cause data to be moved to other shards. It is extremely challenging for an application to continue operating as usual with the underlying data moving around the shards – requiring the code to continuously attempt to adapt to those moving targets. Furthermore, since adding or removing nodes typically involves the change of the sharding-criterion (or a re-hashing), explicit modifications to the application code are in many just unavoidable. In such cases, it would obviously be even more challenging to add or remove nodes non-disruptively.

Management of sharding by the database or by third party solutions, not by the application

Nowadays, some database products can automate the management of some sharding-related operations. Furthermore, some third-party products exist that may be deployed as an additional "coordinator" layer between the application and the database, and that take responsibility over part of the management of sharding operations. Unfortunately, to date none of these approaches have been able to solve any of the core issues that cause all the scale-out sharding related "suffering".