GoshawkDB

GoshawkDB: Programming with Persistent Distributed Objects

Matthew Sackman
matthew@goshawkdb.io

Would you be confident in your answers if someone came and asked you questions about the features and semantics of:

Would you be confident in your answers if someone came and asked you questions about the features and semantics of:

- MySQL/MariaDB?

Would you be confident in your answers if someone came and asked you questions about the features and semantics of:

- MySQL/MariaDB? (Spider / Galera)

Would you be confident in your answers if someone came and asked you questions about the features and semantics of:

- MySQL/MariaDB? (Spider / Galera)
- PostgreSQL?

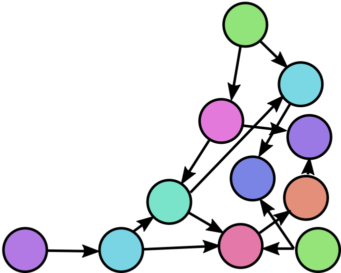
Would you be confident in your answers if someone came and asked you questions about the features and semantics of:

- MySQL/MariaDB? (Spider / Galera)
- PostgreSQL?
- MongoDB?

Would you be confident in your answers if someone came and asked you questions about the features and semantics of:

- MySQL/MariaDB? (Spider / Galera)
- PostgreSQL?
- MongoDB?
- Cassandra?

Part 1: Database Features and Semantics



Desired Features

- Distributed

Desired Features

- Distributed
- Fault-tolerant

Desired Features

- Distributed
- Fault-tolerant
- Automatic sharding

Desired Features

- Distributed
- Fault-tolerant
- Automatic sharding
- Transactional?

Desired Features

- Distributed
- Fault-tolerant
- Automatic sharding
- Transactional?
- Intuitive

Desired Features

- Distributed
- Fault-tolerant
- Automatic sharding
- Transactional?
- Intuitive
- Fast enough

- Distributed:
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

- Distributed: Yes. Primary/Secondaries design; full multi-master with Galera (InnoDB only).
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

- Distributed: Yes. Primary/Secondaries design; full multi-master with Galera (InnoDB only).
- Fault-tolerant: Yes. Galera must be CP; continues working provided a majority of nodes remain connected. Resyncing will happen: expensive.
- Automatic sharding:
- Transactional:
- Intuitive:

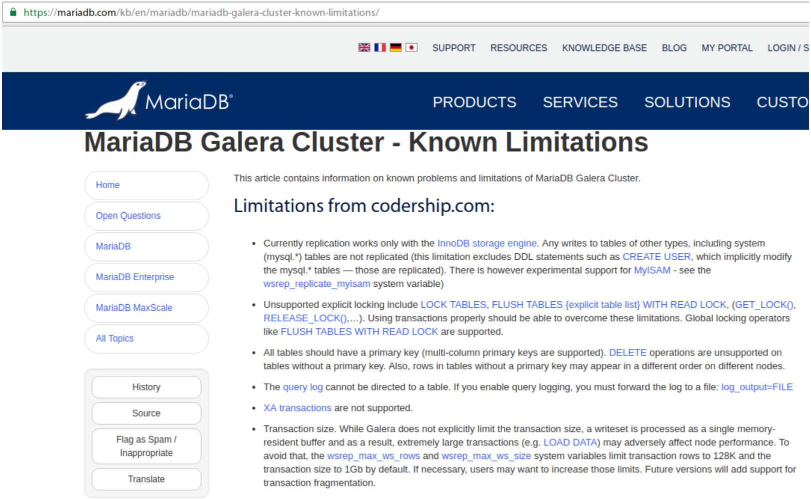
- Distributed: Yes. Primary/Secondaries design; full multi-master with Galera (InnoDB only).
- Fault-tolerant: Yes. Galera must be CP; continues working provided a majority of nodes remain connected. Resyncing will happen: expensive.
- Automatic sharding: Ish. Galera is everyone-has-everything (no sharding). Spider storage engine does do sharding. Spider can be used with Galera.
- Transactional:
- Intuitive:

MariaDB

- Distributed: Yes. Primary/Secondaries design; full multi-master with Galera (InnoDB only).
- Fault-tolerant: Yes. Galera must be CP; continues working provided a majority of nodes remain connected. Resyncing will happen: expensive.
- Automatic sharding: Ish. Galera is everyone-has-everything (no sharding). Spider storage engine does do sharding. Spider can be used with Galera.
- Transactional: Yes, but with Galera, weak isolation levels only: max repeatable read.
- Intuitive:

- Distributed: Yes. Primary/Secondaries design; full multi-master with Galera (InnoDB only).
- Fault-tolerant: Yes. Galera must be CP; continues working provided a majority of nodes remain connected. Resyncing will happen: expensive.
- Automatic sharding: Ish. Galera is everyone-has-everything (no sharding). Spider storage engine does do sharding. Spider can be used with Galera.
- Transactional: Yes, but with Galera, weak isolation levels only: max repeatable read.
- Intuitive: Erm. It's a bit complex!

MariaDB: Galera Limitations



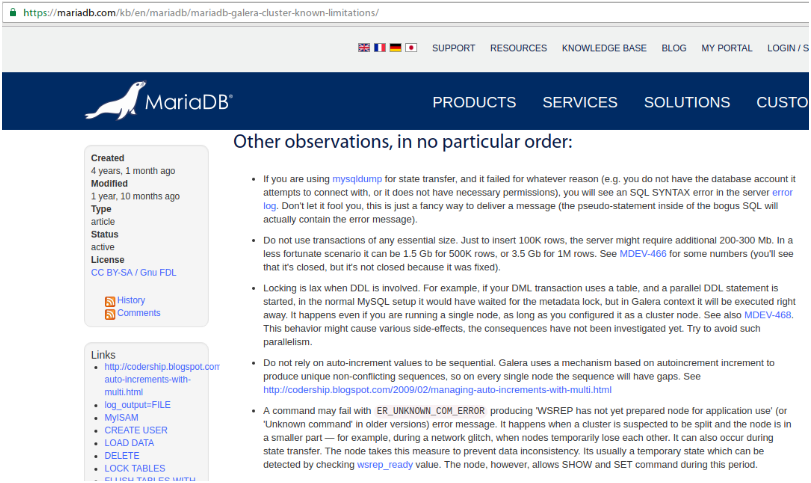
MariaDB Galera Cluster - Known Limitations

This article contains information on known problems and limitations of MariaDB Galera Cluster.

Limitations from codership.com:

- Currently replication works only with the InnoDB storage engine. Any writes to tables of other types, including system (mysql.*) tables are not replicated (this limitation excludes DDL statements such as CREATE USER, which implicitly modify the mysql.* tables — those are replicated). There is however experimental support for MyISAM - see the wsrep_replicate_myisam system variable)
- Unsupported explicit locking include LOCK TABLES, FLUSH TABLES {explicit table list} WITH READ LOCK, (GET_LOCK(), RELEASE_LOCK(),…). Using transactions properly should be able to overcome these limitations. Global locking operators like FLUSH TABLES WITH READ LOCK are supported.
- All tables should have a primary key (multi-column primary keys are supported). DELETE operations are unsupported on tables without a primary key. Also, rows in tables without a primary key may appear in a different order on different nodes.
- The query log cannot be directed to a table. If you enable query logging, you must forward the log to a file: log_output=FILE
- XA transactions are not supported.
- Transaction size. While Galera does not explicitly limit the transaction size, a writeset is processed as a single memory-resident buffer and as a result, extremely large transactions (e.g. LOAD DATA) may adversely affect node performance. To avoid that, the wsrep_max_ws_rows and wsrep_max_ws_size system variables limit transaction rows to 128K and the transaction size to 1Gb by default. If necessary, users may want to increase those limits. Future versions will add support for transaction fragmentation.

MariaDB: Galera Limitations



Created
4 years, 1 month ago
Modified
1 year, 10 months ago
Type
article
Status
active
License
CC BY-SA / Gnu FDL

History
Comments

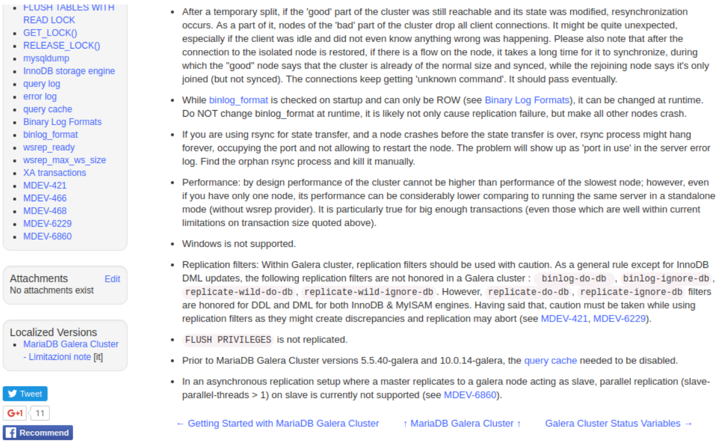
Links

- http://codership.blogspot.com auto-increments-with-multi.html
- log_output=FILE
- MyISAM
- CREATE USER
- LOAD DATA
- DELETE
- LOCK TABLES
- FLUSH TABLES WITH

Other observations, in no particular order:

- If you are using mysqldump for state transfer, and it failed for whatever reason (e.g. you do not have the database account it attempts to connect with, or it does not have necessary permissions), you will see an SQL SYNTAX error in the server error log. Don't let it fool you, this is just a fancy way to deliver a message (the pseudo-statement inside of the bogus SQL will actually contain the error message).
- Do not use transactions of any essential size. Just to insert 100K rows, the server might require additional 200-300 Mb. In a less fortunate scenario it can be 1.5 Gb for 500K rows, or 3.5 Gb for 1M rows. See MDEV-466 for some numbers (you'll see that it's closed, but it's not closed because it was fixed).
- Locking is lax when DDL is involved. For example, if your DML transaction uses a table, and a parallel DDL statement is started, in the normal MySQL setup it would have waited for the metadata lock, but in Galera context it will be executed right away. It happens even if you are running a single node, as long as you configured it as a cluster node. See also MDEV-468. This behavior might cause various side-effects, the consequences have not been investigated yet. Try to avoid such parallelism.
- Do not rely on auto-increment values to be sequential. Galera uses a mechanism based on autoincrement increment to produce unique non-conflicting sequences, so on every single node the sequence will have gaps. See http://codership.blogspot.com/2009/02/managing-auto-increments-with-multi.html
- A command may fail with `ER_UNKNOWN_COM_ERROR` producing 'WSREP has not yet prepared node for application use' (or 'Unknown command' in older versions) error message. It happens when a cluster is suspected to be split and the node is in a smaller part — for example, during a network glitch, when nodes temporarily lose each other. It can also occur during state transfer. The node takes this measure to prevent data inconsistency. Its usually a temporary state which can be detected by checking wsrep_ready value. The node, however, allows SHOW and SET command during this period.

MariaDB: Galera Limitations



- FLUSH TABLES WITH READ LOCK
- GET_LOCK()
- RELEASE_LOCK()
- mysqldump
- InnoDB storage engine
- query log
- error log
- query cache
- Binary Log Formats
- binlog_format
- wsrep_ready
- wsrep_max_ws_size
- XA transactions
- MDEV-421
- MDEV-466
- MDEV-468
- MDEV-6229
- MDEV-6860

Attachments Edit

No attachments exist

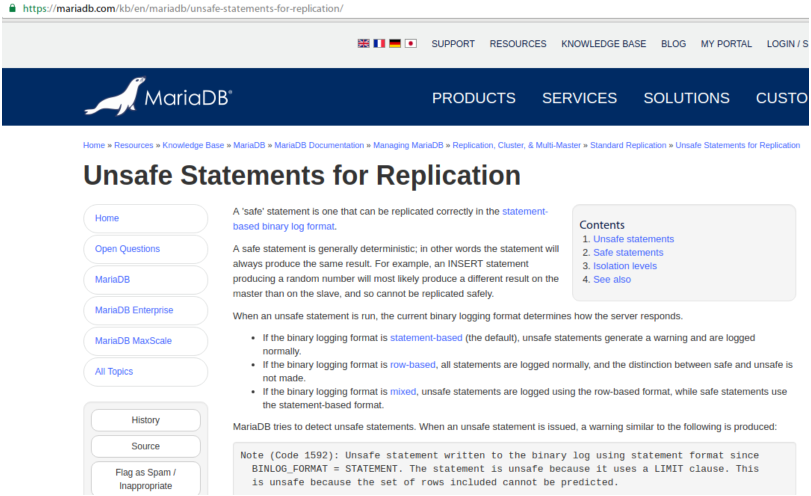
Localized Versions

- MariaDB Galera Cluster - Limitazioni note [it]

- After a temporary split, if the 'good' part of the cluster was still reachable and its state was modified, resynchronization occurs. As a part of it, nodes of the 'bad' part of the cluster drop all client connections. It might be quite unexpected, especially if the client was idle and did not even know anything wrong was happening. Please also note that after the connection to the isolated node is restored, if there is a flow on the node, it takes a long time for it to synchronize, during which the "good" node says that the cluster is already of the normal size and synced, while the rejoining node says it's only joined (but not synced). The connections keep getting 'unknown command'. It should pass eventually.
- While binlog_format is checked on startup and can only be ROW (see Binary Log Formats), it can be changed at runtime. Do NOT change binlog_format at runtime, it is likely not only cause replication failure, but make all other nodes crash.
- If you are using rsync for state transfer, and a node crashes before the state transfer is over, rsync process might hang forever, occupying the port and not allowing to restart the node. The problem will show up as 'port in use' in the server error log. Find the orphan rsync process and kill it manually.
- Performance: by design performance of the cluster cannot be higher than performance of the slowest node; however, even if you have only one node, its performance can be considerably lower comparing to running the same server in a standalone mode (without wsrep provider). It is particularly true for big enough transactions (even those which are well within current limitations on transaction size quoted above).
- Windows is not supported.
- Replication filters: Within Galera cluster, replication filters should be used with caution. As a general rule except for InnoDB DML updates, the following replication filters are not honored in a Galera cluster : `binlog-do-db` , `binlog-ignore-db` , `replicate-wild-do-db` , `replicate-wild-ignore-db` . However, `replicate-do-db` , `replicate-ignore-db` filters are honored for DDL and DML for both InnoDB & MyISAM engines. Having said that, caution must be taken while using replication filters as they might create discrepancies and replication may abort (see MDEV-421, MDEV-6229).
- `FLUSH PRIVILEGES` is not replicated.
- Prior to MariaDB Galera Cluster versions 5.5.40-galera and 10.0.14-galera, the query cache needed to be disabled.
- In an asynchronous replication setup where a master replicates to a galera node acting as slave, parallel replication (slave-parallel-threads > 1) on slave is currently not supported (see MDEV-6860).

← Getting Started with MariaDB Galera Cluster ↑ MariaDB Galera Cluster ↑ Galera Cluster Status Variables →

MariaDB: Unsafe Statements


https://mariadb.com/kb/en/mariadb/unsafe-statements-for-replication/
SUPPORT
RESOURCES
KNOWLEDGE BASE
BLOG
MY PORTAL
LOGIN / S
MariaDB®
PRODUCTS
SERVICES
SOLUTIONS
CUSTO
Home » Resources » Knowledge Base » MariaDB » MariaDB Documentation » Managing MariaDB » Replication, Cluster, & Multi-Master » Standard Replication » Unsafe Statements for Replication
Home
Open Questions
MariaDB
MariaDB Enterprise
MariaDB MaxScale
All Topics
History
Source
Flag as Spam / Inappropriate

Unsafe Statements for Replication

Contents
1. Unsafe statements
2. Safe statements
3. Isolation levels
4. See also

A 'safe' statement is one that can be replicated correctly in the statement-based binary log format.

A safe statement is generally deterministic; in other words the statement will always produce the same result. For example, an INSERT statement producing a random number will most likely produce a different result on the master than on the slave, and so cannot be replicated safely.

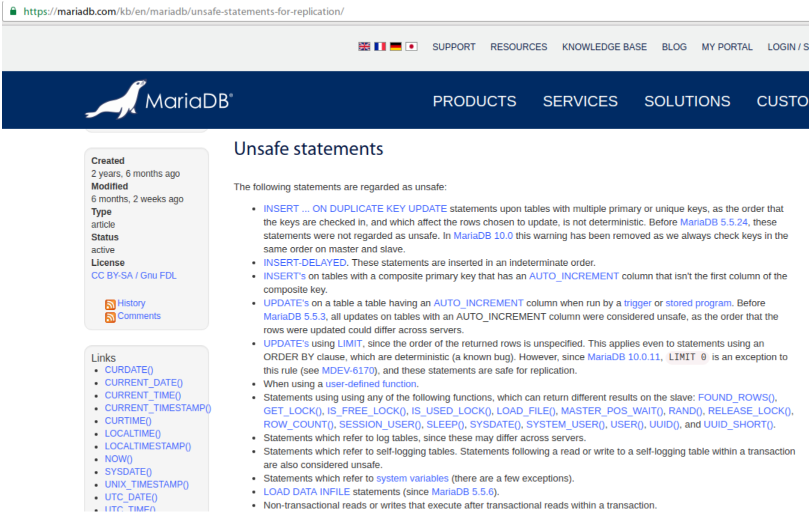
When an unsafe statement is run, the current binary logging format determines how the server responds.

- If the binary logging format is statement-based (the default), unsafe statements generate a warning and are logged normally.
- If the binary logging format is row-based, all statements are logged normally, and the distinction between safe and unsafe is not made.
- If the binary logging format is mixed, unsafe statements are logged using the row-based format, while safe statements use the statement-based format.

MariaDB tries to detect unsafe statements. When an unsafe statement is issued, a warning similar to the following is produced:

```
Note (Code 1592): Unsafe statement written to the binary log using statement format since
  BINLOG_FORMAT = STATEMENT. The statement is unsafe because it uses a LIMIT clause. This
  is unsafe because the set of rows included cannot be predicted.
```

MariaDB: Unsafe Statements



Created
2 years, 6 months ago
Modified
6 months, 2 weeks ago
Type
article
Status
active
License
CC BY-SA / Gnu FDL

History
Comments

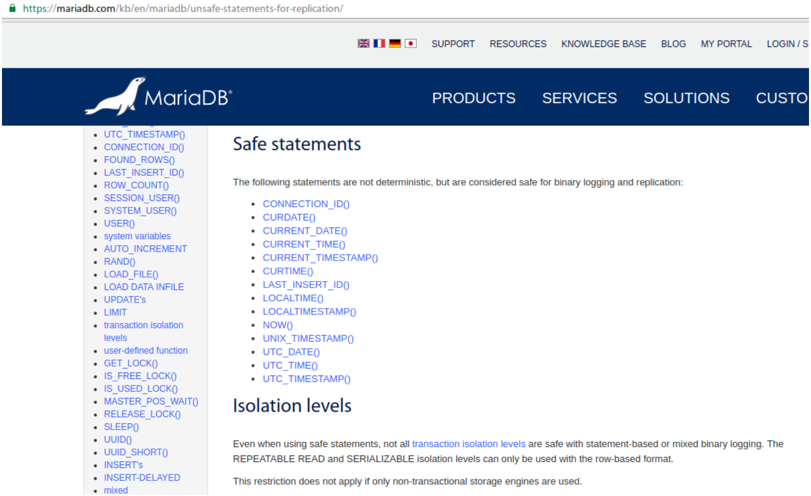
Links
- CURDATE()
- CURRENT_DATE()
- CURRENT_TIME()
- CURRENT_TIMESTAMP()
- CURTIME()
- LOCALTIME()
- LOCALTIMESTAMP()
- NOW()
- SYSDATE()
- UNIX_TIMESTAMP()
- UTC_DATE()
- UTC_TIME()

Unsafe statements

The following statements are regarded as unsafe:

- INSERT ... ON DUPLICATE KEY UPDATE statements upon tables with multiple primary or unique keys, as the order that the keys are checked in, and which affect the rows chosen to update, is not deterministic. Before MariaDB 5.5.24, these statements were not regarded as unsafe. In MariaDB 10.0 this warning has been removed as we always check keys in the same order on master and slave.
- INSERT-DELAYED. These statements are inserted in an indeterminate order.
- INSERT's on tables with a composite primary key that has an AUTO_INCREMENT column that isn't the first column of the composite key.
- UPDATE's on a table a table having an AUTO_INCREMENT column when run by a trigger or stored program. Before MariaDB 5.5.3, all updates on tables with an AUTO_INCREMENT column were considered unsafe, as the order that the rows were updated could differ across servers.
- UPDATE's using LIMIT, since the order of the returned rows is unspecified. This applies even to statements using an ORDER BY clause, which are deterministic (a known bug). However, since MariaDB 10.0.11, `LIMIT 0` is an exception to this rule (see MDEV-6170), and these statements are safe for replication.
- When using a user-defined function.
- Statements using using any of the following functions, which can return different results on the slave: FOUND_ROWS(), GET_LOCK(), IS_FREE_LOCK(), IS_USED_LOCK(), LOAD_FILE(), MASTER_POS_WAIT(), RAND(), RELEASE_LOCK(), ROW_COUNT(), SESSION_USER(), SLEEP(), SYSDATE(), SYSTEM_USER(), USER(), UUID(), and UUID_SHORT().
- Statements which refer to log tables, since these may differ across servers.
- Statements which refer to self-logging tables. Statements following a read or write to a self-logging table within a transaction are also considered unsafe.
- Statements which refer to system variables (there are a few exceptions).
- LOAD DATA INFILE statements (since MariaDB 5.5.6).
- Non-transactional reads or writes that execute after transactional reads within a transaction.

MariaDB: Unsafe Statements



- UTC_TIMESTAMP()
- CONNECTION_ID()
- FOUND_ROWS()
- LAST_INSERT_ID()
- ROW_COUNT()
- SESSION_USER()
- SYSTEM_USER()
- USER()
- system variables
- AUTO_INCREMENT
- RAND()
- LOAD_FILE()
- LOAD DATA INFILE
- UPDATE's
- LIMIT
- transaction isolation levels
- user-defined function
- GET_LOCK()
- IS_FREE_LOCK()
- IS_USED_LOCK()
- MASTER_POS_WAIT()
- RELEASE_LOCK()
- SLEEP()
- UUID()
- UUID_SHORT()
- INSERT's
- INSERT-DELAYED
- mixed

Safe statements

The following statements are not deterministic, but are considered safe for binary logging and replication:

- CONNECTION_ID()
- CURDATE()
- CURRENT_DATE()
- CURRENT_TIME()
- CURRENT_TIMESTAMP()
- CURTIME()
- LAST_INSERT_ID()
- LOCALTIME()
- LOCALTIMESTAMP()
- NOW()
- UNIX_TIMESTAMP()
- UTC_DATE()
- UTC_TIME()
- UTC_TIMESTAMP()

Isolation levels

Even when using safe statements, not all transaction isolation levels are safe with statement-based or mixed binary logging. The REPEATABLE READ and SERIALIZABLE isolation levels can only be used with the row-based format.

This restriction does not apply if only non-transactional storage engines are used.

Stuff that we need to Figure Out

- CP

Stuff that we need to Figure Out

- CP
- Isolation levels (repeatable read)

Stuff that we need to Figure Out

- CP
- Isolation levels (repeatable read)
- How do we decide if we’re violating any of the restrictions?

MongoDB

- Distributed:

- Fault-tolerant:

- Automatic sharding:

- Transactional:
- Intuitive:

- Distributed: Yes. It is a primary/secondaries design so you must connect to the primary to write (and to read!).
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

MongoDB

- Distributed: Yes. It is a primary/secondaries design so you must connect to the primary to write (and to read!).
- Fault-tolerant: Yes. It should survive node failure and recovery. CP system: primary steps down once it loses contact with majority of nodes.
- Automatic sharding:
- Transactional:
- Intuitive:

MongoDB

- Distributed: Yes. It is a primary/secondaries design so you must connect to the primary to write (and to read!).
- Fault-tolerant: Yes. It should survive node failure and recovery. CP system: primary steps down once it loses contact with majority of nodes.
- Automatic sharding: Yes, with several strategies available. Sharding done at collection level.
- Transactional:
- Intuitive:

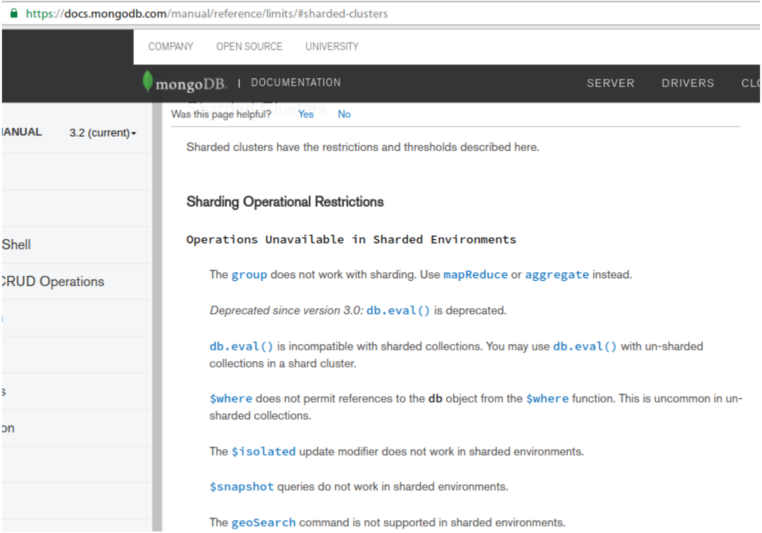
MongoDB

- **Distributed:** Yes. It is a primary/secondaries design so you must connect to the primary to write (and to read!).
- **Fault-tolerant:** Yes. It should survive node failure and recovery. CP system: primary steps down once it loses contact with majority of nodes.
- **Automatic sharding:** Yes, with several strategies available. Sharding done at collection level.
- Transactional: No
- Intuitive:

MongoDB

- **Distributed:** Yes. It is a primary/secondaries design so you must connect to the primary to write (and to read!).
- **Fault-tolerant:** Yes. It should survive node failure and recovery. CP system: primary steps down once it loses contact with majority of nodes.
- **Automatic sharding:** Yes, with several strategies available. Sharding done at collection level.
- **Transactional:** No
- Intuitive: There are some operation issues and restrictions with sharded collections - certain things that no longer work. Also you need to learn *write concerns* and *read concerns*: probably want to set those to *majority* to avoid stale reads.

MongoDB: Sharded Collection Restrictions



Sharded clusters have the restrictions and thresholds described here.

Sharding Operational Restrictions

Operations Unavailable in Sharded Environments

The `group` does not work with sharding. Use `mapReduce` or `aggregate` instead.

Deprecated since version 3.0: `db.eval()` is deprecated.

`db.eval()` is incompatible with sharded collections. You may use `db.eval()` with un-sharded collections in a shard cluster.

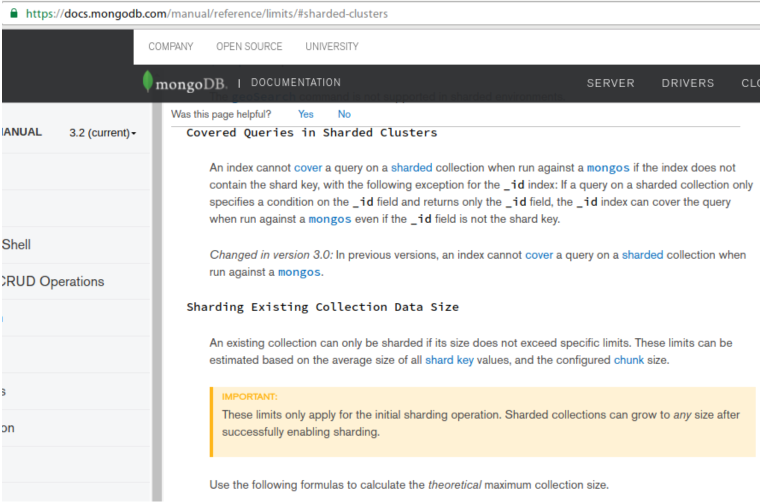
`$where` does not permit references to the **db** object from the `$where` function. This is uncommon in un-sharded collections.

The `$isolated` update modifier does not work in sharded environments.

`$snapshot` queries do not work in sharded environments.

The `geoSearch` command is not supported in sharded environments.

MongoDB: Sharded Collection Restrictions



The geoSearch command is not supported in sharded environments.

Covered Queries in Sharded Clusters

An index cannot cover a query on a sharded collection when run against a mongos if the index does not contain the shard key, with the following exception for the _id index: If a query on a sharded collection only specifies a condition on the _id field and returns only the _id field, the _id index can cover the query when run against a mongos even if the _id field is not the shard key.

Changed in version 3.0: In previous versions, an index cannot cover a query on a sharded collection when run against a mongos.

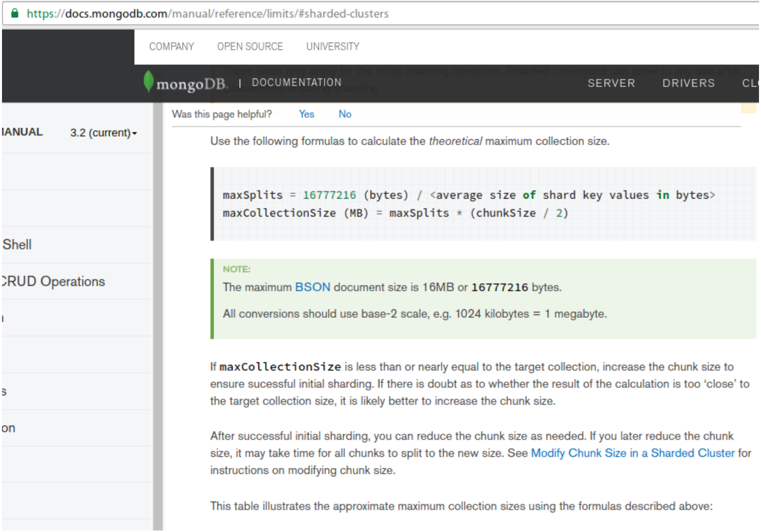
Sharding Existing Collection Data Size

An existing collection can only be sharded if its size does not exceed specific limits. These limits can be estimated based on the average size of all shard key values, and the configured chunk size.

> IMPORTANT:
>
> These limits only apply for the initial sharding operation. Sharded collections can grow to *any* size after successfully enabling sharding.

Use the following formulas to calculate the *theoretical* maximum collection size.

MongoDB: Sharded Collection Restrictions



These limits only apply for the initial sharding operation. Sharded collections can grow to any size after successfully enabling sharding.

Use the following formulas to calculate the *theoretical* maximum collection size.

```
maxSplits = 16777216 (bytes) / <average size of shard key values in bytes>
maxCollectionSize (MB) = maxSplits * (chunkSize / 2)
```

NOTE:

The maximum BSON document size is 16MB or `16777216` bytes.

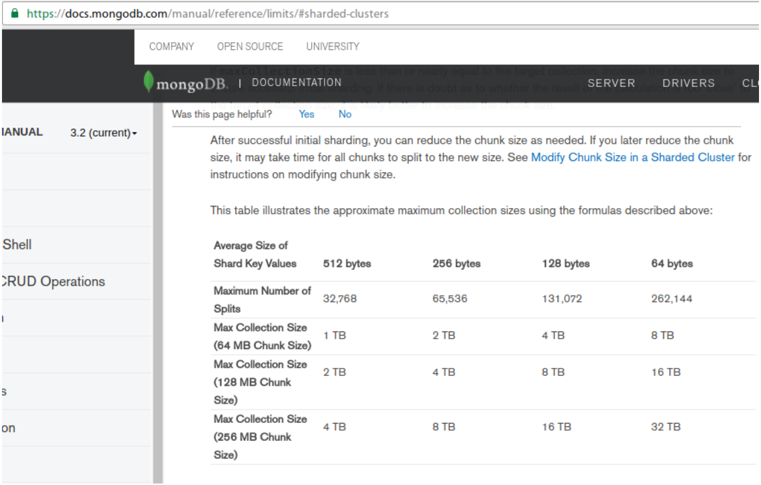
All conversions should use base-2 scale, e.g. 1024 kilobytes = 1 megabyte.

If `maxCollectionSize` is less than or nearly equal to the target collection, increase the chunk size to ensure sucessful initial sharding. If there is doubt as to whether the result of the calculation is too ‘close’ to the target collection size, it is likely better to increase the chunk size.

After successful initial sharding, you can reduce the chunk size as needed. If you later reduce the chunk size, it may take time for all chunks to split to the new size. See Modify Chunk Size in a Sharded Cluster for instructions on modifying chunk size.

This table illustrates the approximate maximum collection sizes using the formulas described above:

MongoDB: Sharded Collection Restrictions



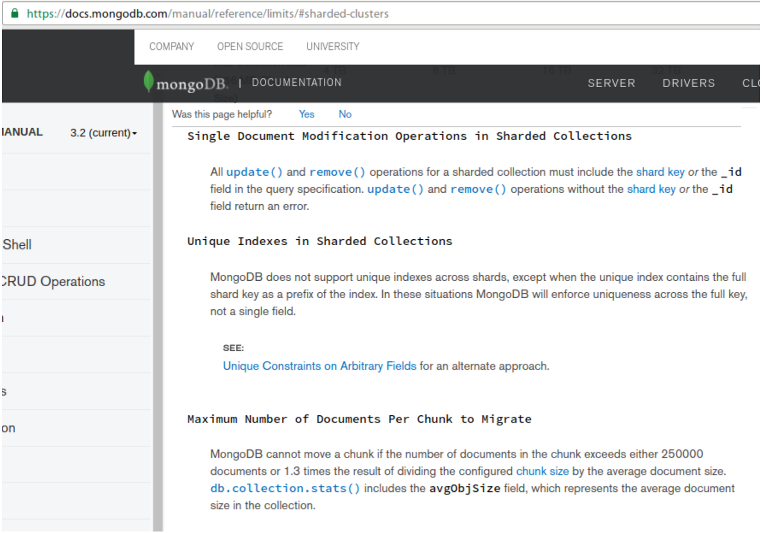
If maxCollectionSize is less than or nearly equal to the target collection, increase the chunk size to ensure successful initial sharding. If there is doubt as to whether the result of the calculation is 'too' close' to the target collection size, it is likely better to increase the chunk size.

After successful initial sharding, you can reduce the chunk size as needed. If you later reduce the chunk size, it may take time for all chunks to split to the new size. See Modify Chunk Size in a Sharded Cluster for instructions on modifying chunk size.

This table illustrates the approximate maximum collection sizes using the formulas described above:

Average Size of Shard Key Values	512 bytes	256 bytes	128 bytes	64 bytes
Maximum Number of Splits	32,768	65,536	131,072	262,144
Max Collection Size (64 MB Chunk Size)	1 TB	2 TB	4 TB	8 TB
Max Collection Size (128 MB Chunk Size)	2 TB	4 TB	8 TB	16 TB
Max Collection Size (256 MB Chunk Size)	4 TB	8 TB	16 TB	32 TB

MongoDB: Sharded Collection Restrictions



Single Document Modification Operations in Sharded Collections

All update() and remove() operations for a sharded collection must include the shard key *or* the _id field in the query specification. update() and remove() operations without the shard key *or* the _id field return an error.

Unique Indexes in Sharded Collections

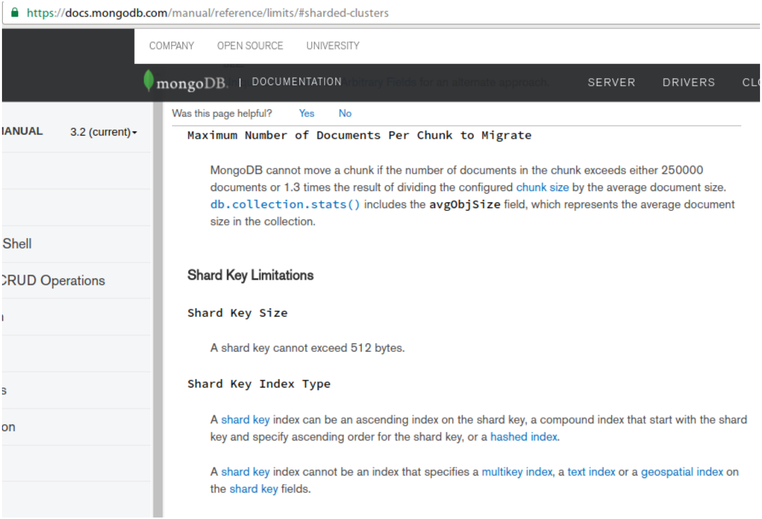
MongoDB does not support unique indexes across shards, except when the unique index contains the full shard key as a prefix of the index. In these situations MongoDB will enforce uniqueness across the full key, not a single field.

SEE:
Unique Constraints on Arbitrary Fields for an alternate approach.

Maximum Number of Documents Per Chunk to Migrate

MongoDB cannot move a chunk if the number of documents in the chunk exceeds either 250000 documents or 1.3 times the result of dividing the configured chunk size by the average document size. db.collection.stats() includes the **avgObjSize** field, which represents the average document size in the collection.

MongoDB: Sharded Collection Restrictions



Maximum Number of Documents Per Chunk to Migrate

MongoDB cannot move a chunk if the number of documents in the chunk exceeds either 250000 documents or 1.3 times the result of dividing the configured chunk size by the average document size. `db.collection.stats()` includes the **`avgObjSize`** field, which represents the average document size in the collection.

Shard Key Limitations

Shard Key Size

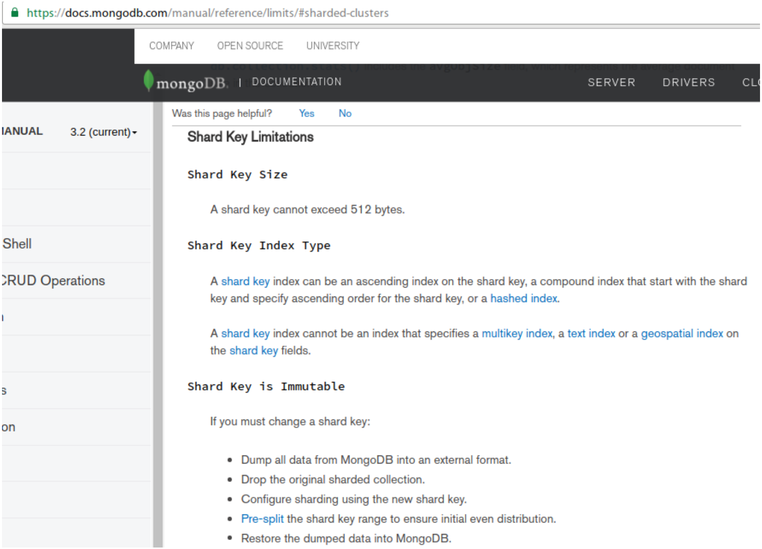
A shard key cannot exceed 512 bytes.

Shard Key Index Type

A shard key index can be an ascending index on the shard key, a compound index that start with the shard key and specify ascending order for the shard key, or a hashed index.

A shard key index cannot be an index that specifies a multikey index, a text index or a geospatial index on the shard key fields.

MongoDB: Sharded Collection Restrictions



Shard Key Limitations

Shard Key Size

A shard key cannot exceed 512 bytes.

Shard Key Index Type

A shard key index can be an ascending index on the shard key, a compound index that start with the shard key and specify ascending order for the shard key, or a hashed index.

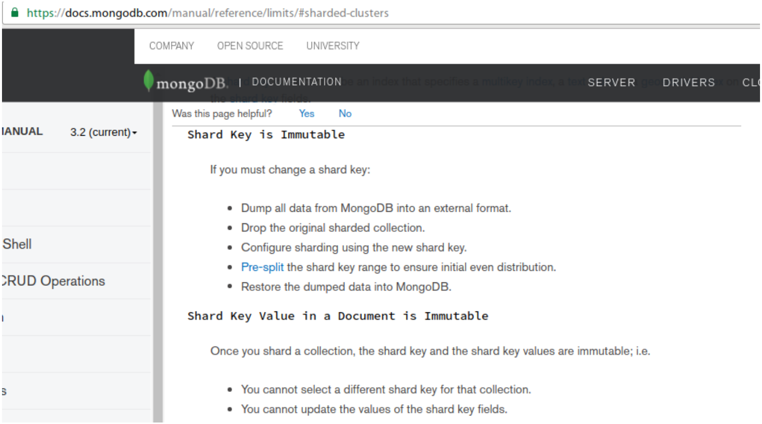
A shard key index cannot be an index that specifies a multikey index, a text index or a geospatial index on the shard key fields.

Shard Key is Immutable

If you must change a shard key:

- Dump all data from MongoDB into an external format.
- Drop the original sharded collection.
- Configure sharding using the new shard key.
- Pre-split the shard key range to ensure initial even distribution.
- Restore the dumped data into MongoDB.

MongoDB: Sharded Collection Restrictions



Shard Key is Immutable

If you must change a shard key:

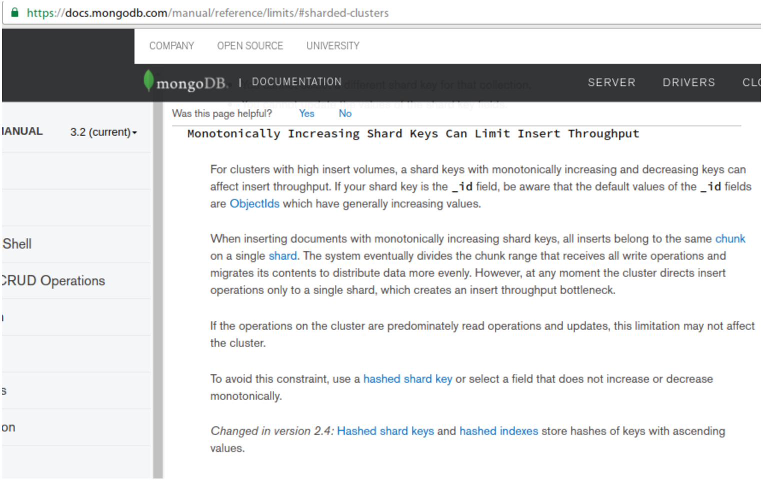
- Dump all data from MongoDB into an external format.
- Drop the original sharded collection.
- Configure sharding using the new shard key.
- Pre-split the shard key range to ensure initial even distribution.
- Restore the dumped data into MongoDB.

Shard Key Value in a Document is Immutable

Once you shard a collection, the shard key and the shard key values are immutable; i.e.

- You cannot select a different shard key for that collection.
- You cannot update the values of the shard key fields.

MongoDB: Sharded Collection Restrictions


https://docs.mongodb.com/manual/reference/limits/#sharded-clusters
COMPANY
OPEN SOURCE
UNIVERSITY
mongoDB.
DOCUMENTATION
SERVER
DRIVERS
CL
You cannot select a different shard key for that collection.
You cannot update the values of the shard key fields.
Was this page helpful?
Yes
No
IANUAL
3.2 (current)
Shell
CRUD Operations
ı
s
on

`Monotonically Increasing Shard Keys Can Limit Insert Throughput`

For clusters with high insert volumes, a shard keys with monotonically increasing and decreasing keys can affect insert throughput. If your shard key is the `_id` field, be aware that the default values of the `_id` fields are ObjectIds which have generally increasing values.

When inserting documents with monotonically increasing shard keys, all inserts belong to the same chunk on a single shard. The system eventually divides the chunk range that receives all write operations and migrates its contents to distribute data more evenly. However, at any moment the cluster directs insert operations only to a single shard, which creates an insert throughput bottleneck.

If the operations on the cluster are predominately read operations and updates, this limitation may not affect the cluster.

To avoid this constraint, use a hashed shard key or select a field that does not increase or decrease monotonically.

Changed in version 2.4: Hashed shard keys and hashed indexes store hashes of keys with ascending values.

Stuff that we need to Figure Out

- CP
- Isolation levels (repeatable read)
- How do we decide if we're violating any of the restrictions?

Stuff that we need to Figure Out

- CP
- Isolation levels (repeatable read)
- How do we decide if we're violating any of the restrictions?
- Write-concern, Read-concern, read-preference

Cassandra

- Distributed:
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

Cassandra

- Distributed: Yes. Logically available from any node.
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

Cassandra

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. Replication factor which is the 2F+1 value. Hinted Handoff to store write hints if the write concern is lower than the replication factor. It's LWW with L defined by some timestamp…
- Automatic sharding:
- Transactional:
- Intuitive:

Cassandra

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. Replication factor which is the 2F+1 value. Hinted Handoff to store write hints if the write concern is lower than the replication factor. It's LWW with L defined by some timestamp…
- Automatic sharding: Yes. Based on consistent-hash ring.
- Transactional:
- Intuitive:

Cassandra

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. Replication factor which is the 2F+1 value. Hinted Handoff to store write hints if the write concern is lower than the replication factor. It's LWW with L defined by some timestamp…
- Automatic sharding: Yes. Based on consistent-hash ring.
- Transactional: No. Supports *light weight transactions* which only work on single objects. Question marks about current implementation.
- Intuitive:

Cassandra

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. Replication factor which is the 2F+1 value. Hinted Handoff to store write hints if the write concern is lower than the replication factor. It's LWW with L defined by some timestamp…
- Automatic sharding: Yes. Based on consistent-hash ring.
- Transactional: No. Supports *light weight transactions* which only work on single objects. Question marks about current implementation.
- Intuitive: AP with write consistency level *ANY* and *RF*=|*nodes*| and *read*=*ONE*. CP with *light weight transactions* and use of *serial* consistency, which achieves linearizable isolation, which is consistent with serial of LWW. Internally uses Paxos, but has to start at phase 1. Docs suggest 4 round trips…which seems like 2 too many. Many things between AP and CP are possible too.

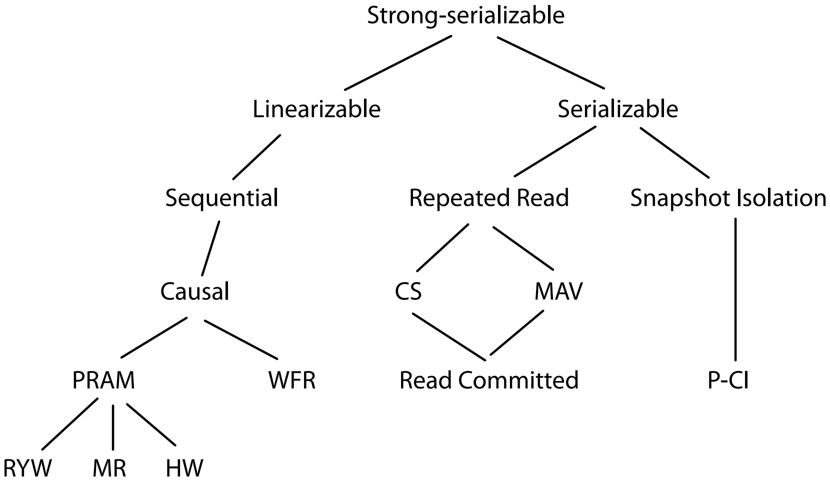
Stuff that we need to Figure Out

- CP
- Isolation levels (repeatable read)
- How do we decide if we're violating any of the restrictions?
- Write-concern, Read-concern, read-preference

Stuff that we need to Figure Out

- CP and AP: CAP "theorem"
- Isolation levels (repeatable read)
- How do we decide if we're violating any of the restrictions?
- Write-concern, Read-concern, read-preference
- LWW
- Timestamps and Clocks
- Consistent-hash
- Paxos

Isolation Levels



Snapshot Isolation

as per Wikipedia

“Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.”

Snapshot Isolation

as per Wikipedia

“Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.”

Snapshot isolation is called “serializable” mode in Oracle.

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:
  `t1` then `t2`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  t1 then t2: `x:0, y:1`

Snapshot Isolation

trouble up mill

```
x, y := 0,0

func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`: `x:1, y:0`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  t1 then t2: x:0, y:1

  t2 then t1: x:1, y:0

- Snapshot Isolation:

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`: `x:1, y:0`

- Snapshot Isolation:

  `t1` || `t2`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  t1 then t2: x:0, y:1

  t2 then t1: x:1, y:0

- Snapshot Isolation:

  t1 || t2

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`
  `t2` then `t1`: `x:1, y:0`

- Snapshot Isolation:

  `t1` || `t2`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`:   `x:0, y:1`

  `t2` then `t1`:   `x:1, y:0`

- Snapshot Isolation:

  `t1` || `t2`:   `x:1, y:1`

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  t1 then t2: x:0, y:1

  t2 then t1: x:1, y:0

- Snapshot Isolation: Write Skew

  t1 || t2: x:1, y:1

Serialized versus Strong Serialized

Strong Serialized must obey causality. Serialized does not need to.

Serialized

```
x := 0

func t1() {
  if x == 0 {
    x = 1
  }
}
func t2() {
  if x == 1 {
    x = 2
  }
}
```

Serialized

```
x := 0

func t1() {
  if x == 0 {
    x = 1
  }
}
func t2() {
  if x == 1 {
    x = 2
  }
}
```

- Client runs: `t1; t2;`

Serialized

```
x := 0

func t1() {
  if x == 0 {
    x = 1
  }
}
func t2() {
  if x == 1 {
    x = 2
  }
}
```

- Client runs: `t1; t2;`
- Server is allowed to reorder though: `t1; t2;` or `t2; t1;`

Snapshot Isolation

as per Wikipedia

“Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.”

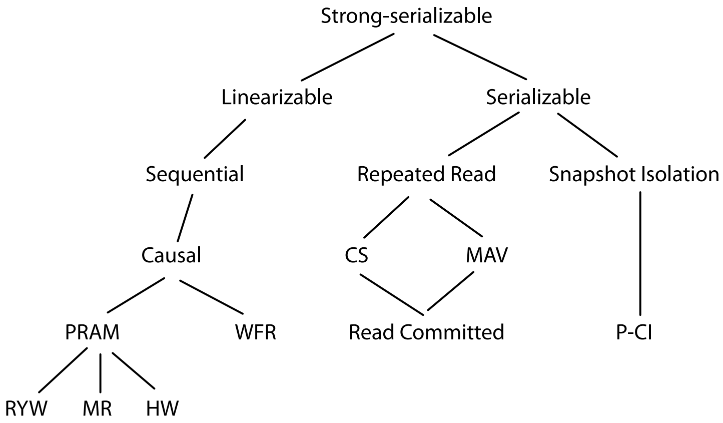
Snapshot Isolation

as per Wikipedia

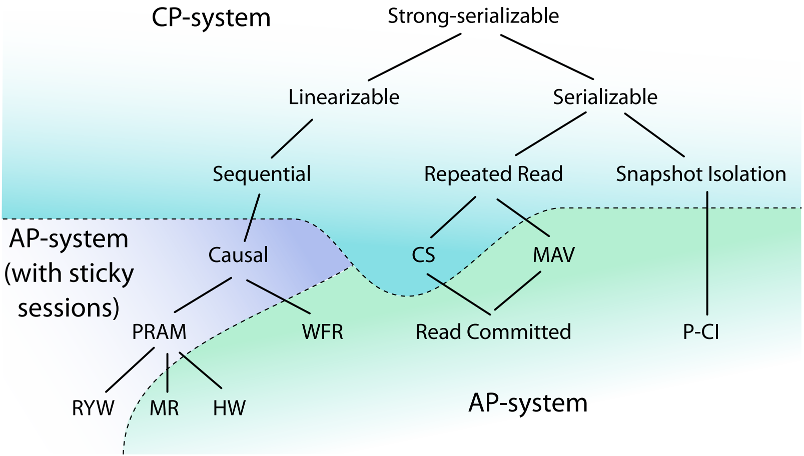
“Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.”

As with serializable, no restriction on *when* in the history of the database each snapshot is taken, so again can violate causality.

Isolation Levels



Isolation Levels



GoshawkDB

- Distributed:
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

- Distributed: Yes. Logically available from any node.
- Fault-tolerant:
- Automatic sharding:
- Transactional:
- Intuitive:

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. You specify the number of failures you wish to withstand: F.
- Automatic sharding:
- Transactional:
- Intuitive:

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. You specify the number of failures you wish to withstand: *F*.
- Automatic sharding: Yes. Completely transparent.
- Transactional:
- Intuitive:

- Distributed: Yes. Logically available from any node.
- Fault-tolerant: Yes. You specify the number of failures you wish to withstand: *F*.
- Automatic sharding: Yes. Completely transparent.
- Transactional: Yes. Strong serializable only.
- Intuitive:

- **Distributed:** Yes. Logically available from any node.
- **Fault-tolerant:** Yes. You specify the number of failures you wish to withstand: F.
- **Automatic sharding:** Yes. Completely transparent.
- **Transactional:** Yes. Strong serializable only.
- Intuitive: Hopefully! Small API, small featureset, clear docs.

Part 1: Conclusions

- Understanding databases is hard: lots of terminology

Part 1: Conclusions

- Understanding databases is hard: lots of terminology
- Comparing them is harder

Part 1: Conclusions

- Understanding databases is hard: lots of terminology
- Comparing them is harder
- Very common that your requirements grow over time

Part 1: Conclusions

- Understanding databases is hard: lots of terminology
- Comparing them is harder
- Very common that your requirements grow over time
- Hedge your bets: go with something that offers strong guarantees and simple intuitive semantics…

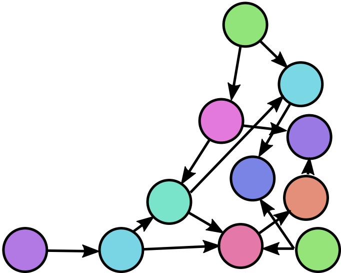
Part 1: Conclusions

- Understanding databases is hard: lots of terminology
- Comparing them is harder
- Very common that your requirements grow over time
- Hedge your bets: go with something that offers strong guarantees and simple intuitive semantics…
- …assuming it's fast enough. The stronger the guarantees, the more work that has to be done.

Part 1: Conclusions

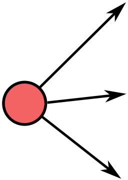
- Understanding databases is hard: lots of terminology
- Comparing them is harder
- Very common that your requirements grow over time
- Hedge your bets: go with something that offers strong guarantees and simple intuitive semantics…
- …assuming it's fast enough. The stronger the guarantees, the more work that has to be done.
- At scale, problems stop being rare.

Part 2: APIs

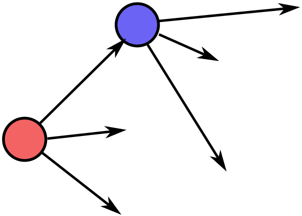


GoshawkDB and the Object Graph

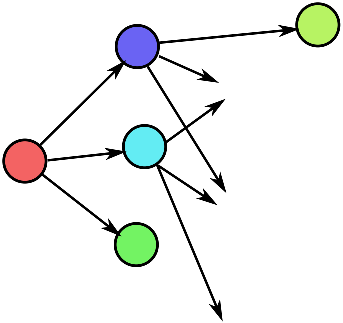
GoshawkDB and the Object Graph



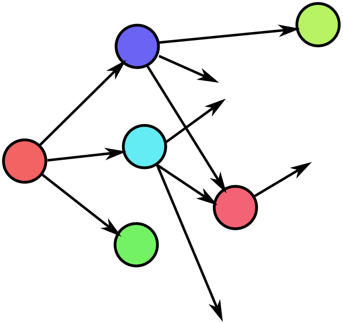
GoshawkDB and the Object Graph



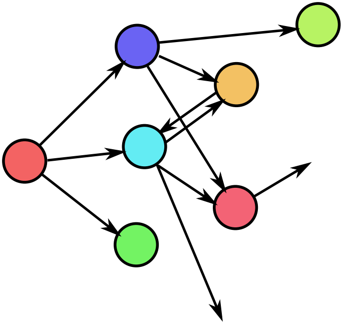
GoshawkDB and the Object Graph



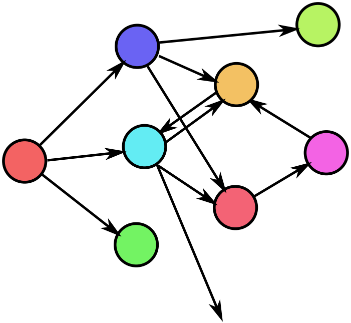
GoshawkDB and the Object Graph



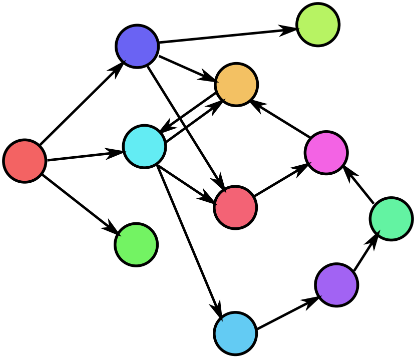
GoshawkDB and the Object Graph



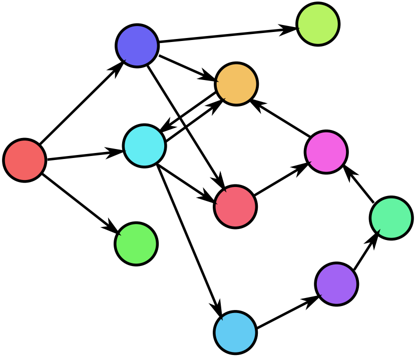
GoshawkDB and the Object Graph



GoshawkDB and the Object Graph



GoshawkDB and the Object Graph



You deal with objects in your programming language. Why not make them persistent?

A Perfect World

Imagine infinite RAM and CPU, and no crashes.
How would we write programs and manage data?

A Perfect World

Imagine infinite RAM and CPU, and no crashes.
How would we write programs and manage data?
It depends.

Object Relational Mappings

- Tend to do one table per class/type…

Object Relational Mappings

- Tend to do one table per class/type…
- …which can create unnecessary contention.

Object Relational Mappings

- Tend to do one table per class/type…
- …which can create unnecessary contention.
- Always introduce tension as to who writes the SQL

Object Relational Mappings

- Tend to do one table per class/type…
- …which can create unnecessary contention.
- Always introduce tension as to who writes the SQL
- Often produce inefficient SQL

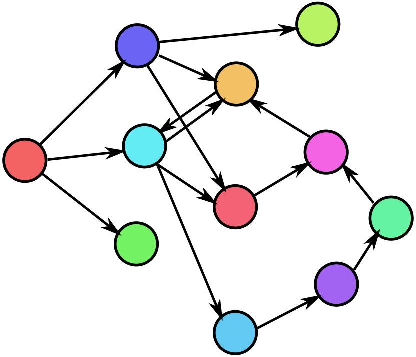
Object Relational Mappings

- Tend to do one table per class/type…
- …which can create unnecessary contention.
- Always introduce tension as to who writes the SQL
- Often produce inefficient SQL
- Sometimes do weird and wacky things to your schema

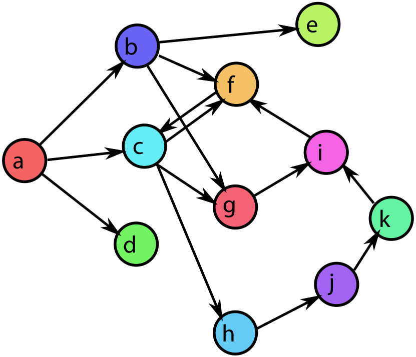
Object Relational Mappings

- Tend to do one table per class/type…
- …which can create unnecessary contention.
- Always introduce tension as to who writes the SQL
- Often produce inefficient SQL
- Sometimes do weird and wacky things to your schema
- Introduce another layer of complexity; more code, more dependencies

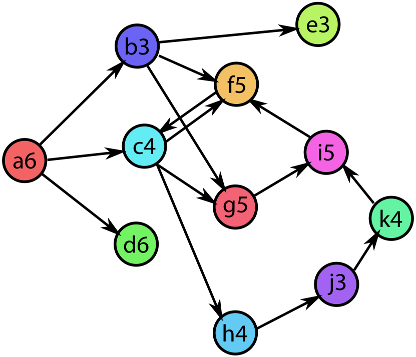
Object Graph with MVCC



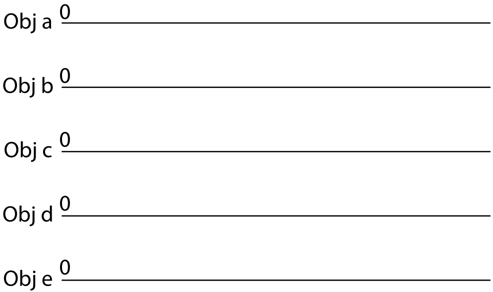
Object Graph with MVCC



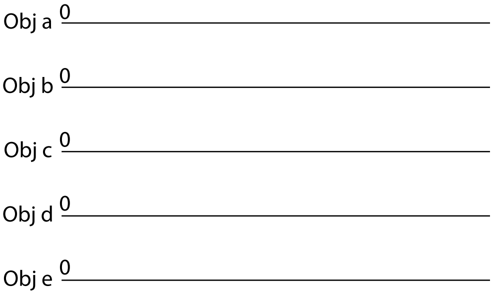
Object Graph with MVCC



MVCC Example



MVCC Example



Obj a 0

Obj b 0

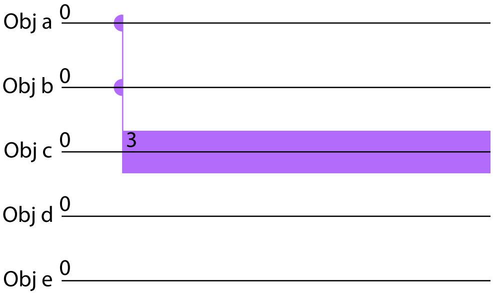
Obj c 0

Obj d 0

Obj e 0

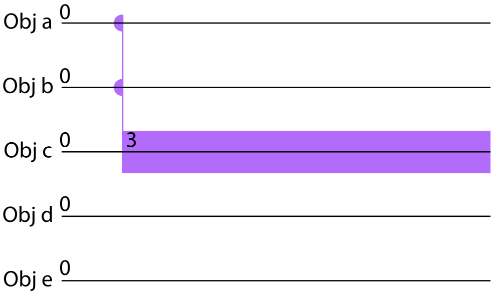
txn3 r[a0,b0]w[c]

MVCC Example



txn3 r[a0,b0]w[c]

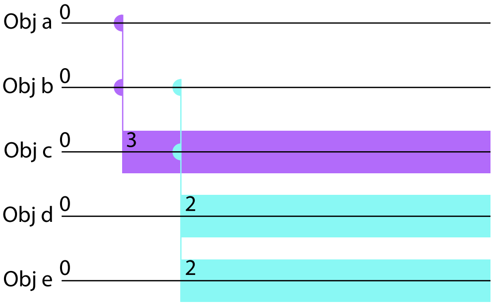
MVCC Example



txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

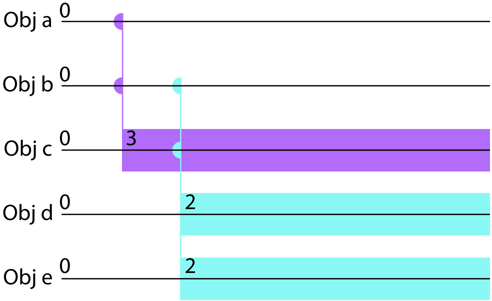
MVCC Example



txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

MVCC Example

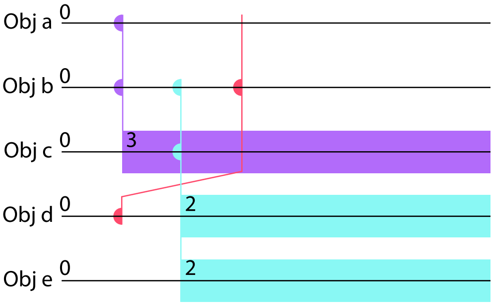


txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

txn7 r[b0,d0]w[a,c]

MVCC Example

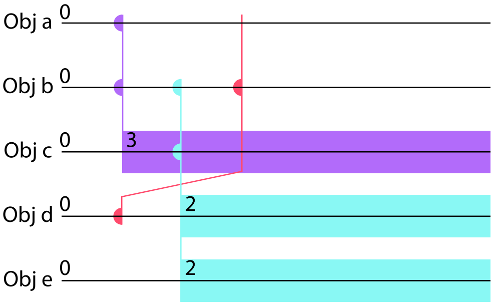


txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

txn7 r[b0,d0]w[a,c]

MVCC Example

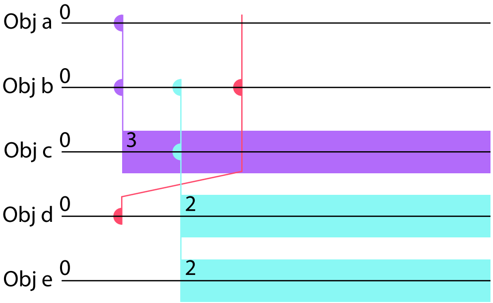


txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

~~txn7 r[b0,d0]w[a,c]~~

MVCC Example



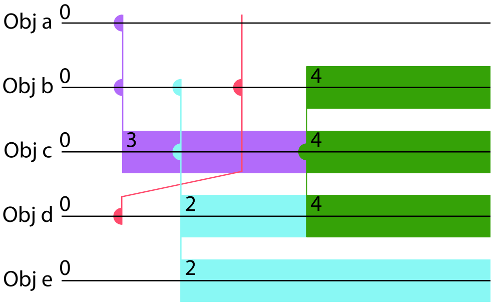
txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

~~txn7 r[b0,d0]w[a,c]~~

txn4 r[c3]w[b,c,d]

MVCC Example



txn3 r[a0,b0]w[c]

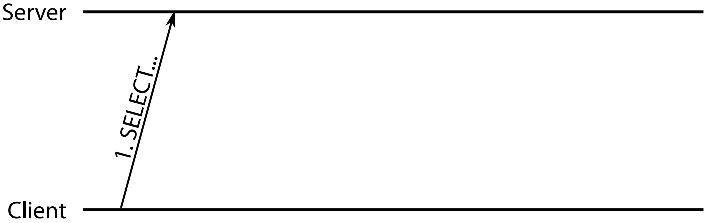
txn2 r[b0,c3]w[d,e]

~~txn7 r[b0,d0]w[a,c]~~

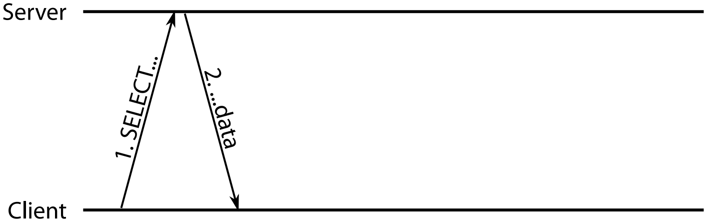
txn4 r[c3]w[b,c,d]

Txn Lifecycle and the Network

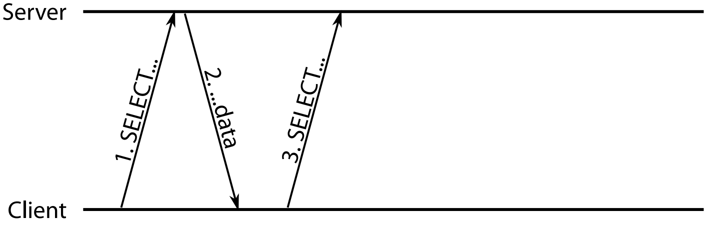
Txn Lifecycle and the Network



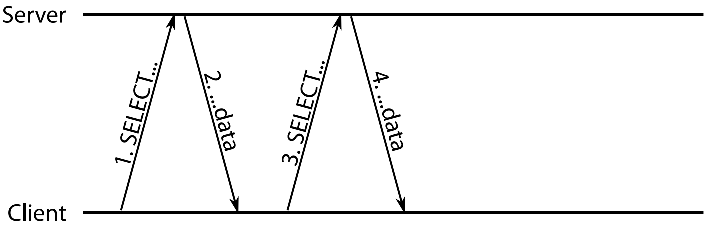
Txn Lifecycle and the Network



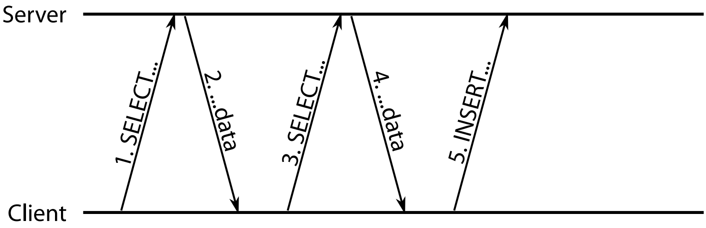
Txn Lifecycle and the Network



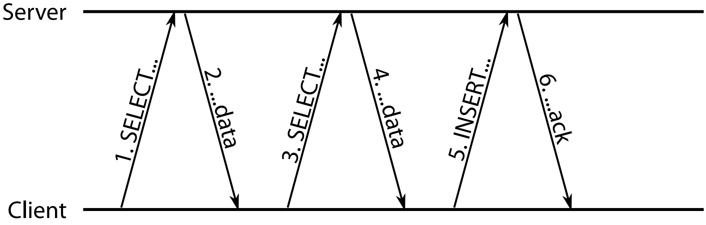
Txn Lifecycle and the Network



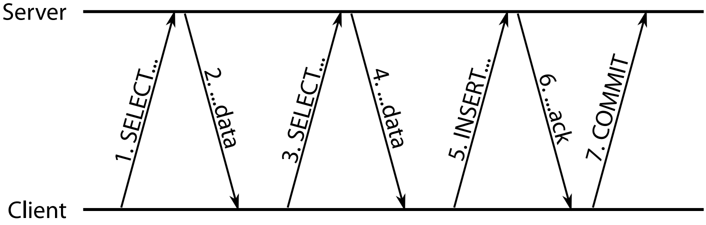
Txn Lifecycle and the Network



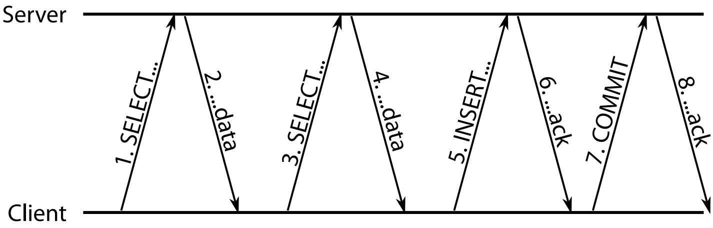
Txn Lifecycle and the Network



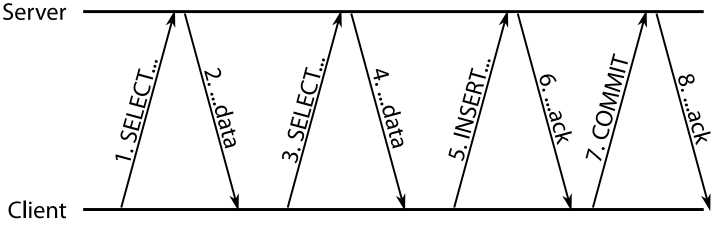
Txn Lifecycle and the Network



Txn Lifecycle and the Network



Txn Lifecycle and the Network



Consequently, SQL has lots of commands (which raises complexity), in order to reduce number of round-trips.

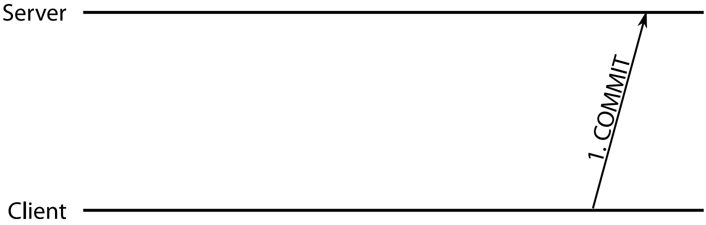
Also stored-procedures.

Txn Lifecycle and the Network

Assume client-side hot cache

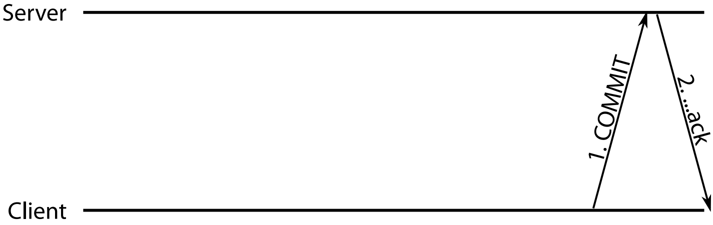
Txn Lifecycle and the Network

Assume client-side hot cache



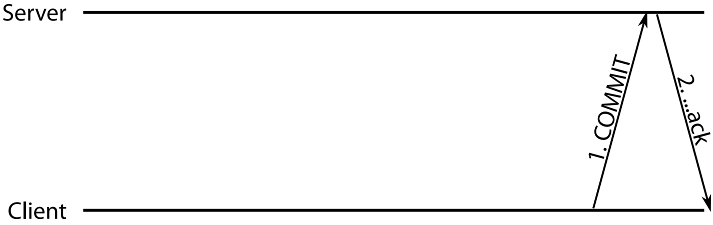
Txn Lifecycle and the Network

Assume client-side hot cache



Txn Lifecycle and the Network

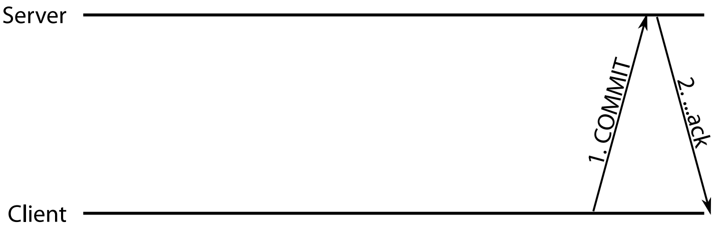
Assume client-side hot cache



Traditional database: the client asks “what is the value of x?”

Txn Lifecycle and the Network

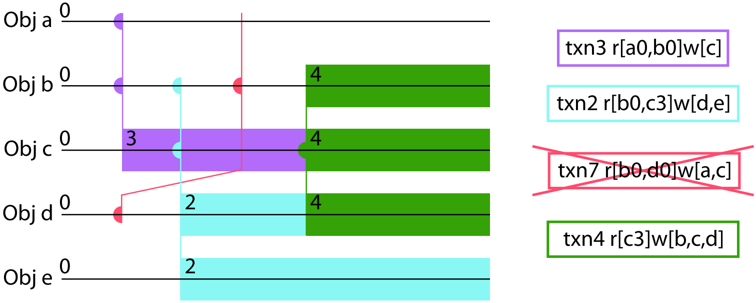
Assume client-side hot cache



Traditional database: the client asks “what is the value of x?”
GoshawkDB: the client asks “are these reads and writes consistent with the current state of the data?”

Txn Lifecycle and the Network

Assume client-side hot cache



txn3 r[a0,b0]w[c]

txn2 r[b0,c3]w[d,e]

~~txn7 r[b0,d0]w[a,c]~~

txn4 r[c3]w[b,c,d]

Traditional database: the client asks “what is the value of x?”
GoshawkDB: the client asks “are these reads and writes consistent with the current state of the data?”

Ship all the Objects to the Client?

- Client loads objects on demand by navigating from a root object

Ship all the Objects to the Client?

- Client loads objects on demand by navigating from a root object
- As usual, use data structures that allow efficient access to large numbers of objects

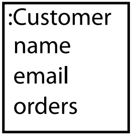
Ship all the Objects to the Client?

- Client loads objects on demand by navigating from a root object
- As usual, use data structures that allow efficient access to large numbers of objects
- In programming languages, eg HashMap

Ship all the Objects to the Client?

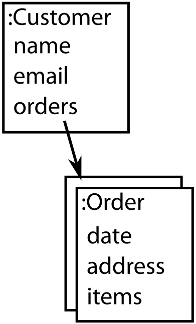
- Client loads objects on demand by navigating from a root object
- As usual, use data structures that allow efficient access to large numbers of objects
- In programming languages, eg HashMap
- In databases, some sort of Index

Shopping

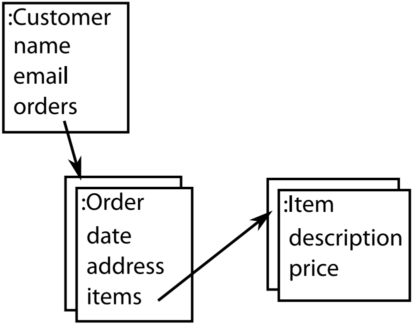


:Customer
name
email
orders

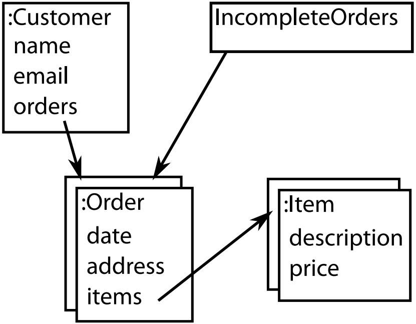
Shopping



Shopping


:Customer
name
email
orders
:Order
date
address
items
:Item
description
price

Shopping



:Customer
name
email
orders

IncompleteOrders

:Order
date
address
items

:Item
description
price

Conclusions

GoshawkDB

- Databases have a habit of *going wrong* once everything else is on fire
- The easier it is to understand the semantics of the database, the less you'll be surprised by it
- We don't always need tables, or query languages
- GoshawkDB: distributed, fault tolerant, transactional, object store

https://goshawkdb.io/