



# Scalability Considerations

Navid Malek  
DevOps Engineer - Nopayar  
[navidmalekedu@gmail.com](mailto:navidmalekedu@gmail.com)



# Scaling Hierarchy

[Reference article](#)



# Three main tools

- Caching and Buffering
- Replication
- Functional Partitioning and Sharding



# Considerations

“Scale-UP” is limited and expensive

- Especially when it comes to “single thread” performance

Simple next choices:

- Caching, Buffering, Queuing
- Functional Partition
- Replication



# Considerations

Best way to optimize something is stop doing it

- Make sure you only get the data you need.

If you can't avoid doing it cache it

In any case consider batch it/buffer it

- Round trips are expensive

If possible delay/do it in the background

- Queueing

A lot can be done in addition to Optimizing operation!



# Considerations

The highest performance/scalability often comes with increased complexity

- But you can have slow AND complex too

Cache hit on highest level is best

Kind of caches

- Browser cache
- Squid Cache
- Memcache
- Database buffer cache



# Caching policies

## Time Based

- This item expires in 10 minutes.
- Easy to implement. Few objects cachable

## Write Invalidate

- Changes to object invalidate dependent cache entries

## Cache Update

- Changes to object cause update dependent cache entries

## Version based Caching

- Check actual object version vs version in cache.



# Where to cache

## Browser Cache

- TTL Based

## Squid,Varnish

- TTL, eTag, Simple invalidation

## Memcache

- TTL, Invalidation, Update, Checking version

## APC/Local in process cache

- TTL, version based, some invalidation





# Memcached vs Redis

Both Memcached and Redis serve as in-memory, key-value data stores, although Redis is more accurately described as a [data structure store](#).

Redis is more powerful, more popular, and better supported than memcached. Memcached can only do a small fraction of the things Redis can do. Redis is better even where their features overlap.

For anything new, use Redis.



# Batching

Less round trips is always good

Think “Set at once” vs Object at once

- `get_messages()` vs `get_message`

Set API allows to parallelize operations

- `curl_multi` to fetch multiple URLs in parallel

Goes hand in hand with buffering/queuing

There is optimal batch size

- After which diminishing returns or performance loss



# Buffering

May be similar to batching

- Accumulate a lot of changes do them at once

Also could aggregate changes, doing one instead of many

- Counters are good example

Buffering can be done

- Inside process; File; Memcache; MEMORY/MyISAM table; Redis etc
- Pick best depending on type of data and durability requirement



# Queuing

Doing work in background

- Helpful for complex operations

Using together with Buffering for better Perf.

Message Delivery

- Cross data center network is slow and unreliable

Generally convenient programming concept

- Such as Job Management

## Software for Queueing

A lot to chose from depending on needs !

Simple Files

MySQL Table

Q4M (MySQL Storage Engine)

Redis

Gearman

RabbitMQ

ActiveMQ

# Background work

Two types of work for User Interaction


- Required to Generate response (synchronous)
- Work which does NOT need to happen at once

## User Experience design Question

- Credit card charge confirmed vs Queued to be processed
- Report instantly shown vs Generated in background

## Background activities are best for performance

- A lot more flexible load management
- Need to ensure behavior does not annoy user.



Instagram  
Background  
Data Prefetching  
Experiences



# Intelligent queuing and buffering

Multiple tasks in the queue

Higher load = larger queue

Larger queue = better optimization opportunities

Intelligent queue processing

- Picking all tasks related to one object and processing them at once
- Process all reports for given user at once. Better hit rate



# Distributing Data

## Replication

- Multiple data copies.

### Scaling Reads

## Functional Partitioning

- Splitting data by function.

### Scale reads and writes

## “Sharding”

- Horizontal partition. Different users on different nodes.

### Scale reads and writes.



# Functional Partitioning

“Let me put forums database on different MySQL Server”

- Picking set of tables which are mostly independent from the other MySQL instances
- Light duty joins can be coded in application
- These vertical partitions tend to grow too large.





# Fault tolerant

Replication/DRBD/etc to keep component available

Designing application not to fail if single component does not work

No need for all web site to be down if forums are unavailable

- Even if last forum messages featured on the front page

Design application to restrict functionality rather than fail.



# MySQL Replication

Many applications have mostly read load

- Though most of those reads are often served from Memcache or other cache

Using one or several slaves to assist with read load

MySQL Replication is asynchronous

- Special care needed to avoid reading stale data

Does not help to scale writes

- Slaves have lower write capacity than master because

they execute queries in single thread, and writes are duplicated on every slave

Slave caches is typically highly duplicated.



# Taking care of Async Replication

## Query based

- Use Slave for reporting queries

## Session Based

- User which did not modify data can read stale data
- Store binlog position when modification was made

## Data Version/Time based

- User was not modified today – read all his blog posts from the slave

## MySQL Proxy Based

- Work is being done to automatically route queries to slave if they can use it



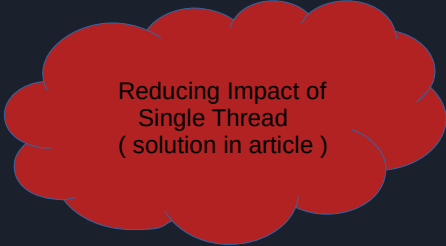
# Replication And Writes

Very fast degradation

- Master 50% busy with writes. 2 Slaves have 50% room for read queries
- 1 “Server Equivalent” capacity for the slaves
- Master load grows 50% and it becomes 75% busy. There is 25% room on each of slaves
- Both Slaves are now equivalent to  $\frac{1}{2}$  of “Server Equivalent”

Single Thread Bottleneck ( even worse because of fairness in kernel)

- Use single CPU
- Submit single IO request at the time (most of the time)
- Getting more common as servers get more cores



Reducing Impact of  
Single Thread  
( solution in article )



# Galera Cluster

- **Synchronous Replication** No slave lag, no data is lost at node crash.
- **Tightly Coupled** All nodes hold the same state. No diverged data between nodes allowed.
- **Multi-threaded Slave** For better performance. For any workload.

[Referenced from galera website](#)



# Different Schema

You can have Different Schema on Master and Slave

- Use extreme care using this. You've been warned ( often makes query routing more complicated and leads to disaster )

Different indexes on Master and Slaves

- Query mix can be different ( E.g. optimization for read queries in slaves )

Different Partitioning settings

Different Storage Engines

Extra columns on the slave

- For example containing cache



# Slave backup and replication

How to easily clone slave from the master ?

- LVM snapshot
- Xtrabackup/InnoDB Hot Backup for InnoDB only.

Replication can run out of sync

- Writing to the wrong slave from Application
- Operational Errors
- MySQL Replication bugs
- Master/Slave Crashes

Validate replication consistency regularly  
– **Mk-table-checksum** is a great online tool



# Sharding

Can be hard to implement

- Especially if application is not designed with sharding in mind

You may need to “JOIN” data with some global tables

- User information, regions, countries etc

Just join things Manually ( at application )

- Also makes caching these items more efficient ( with query separation )

Replication of global tables

- Could be MySQL replication or copy for constant tables.





# Pinterest Best Practices

[Reference video](#)



## Core technology

Amazon EC2 + S3 + ELB, Akamai

Page Views / Day

90+ Web Engines + 50 API Engines

66 MySQL DBs (m1.xlarge) + 1 slave each

59 Redis Instances

51 Memcache Instances

1 Redis Task Manager + 25 Task Processors

Jan 2011

Sharded Solr

## Clustering vs Sharding

Distributes data across nodes automatically	Distributes data across nodes manually
Data can move	Data does not move
Rebalances to distribute load	Split data to distribute load
Nodes communicate with each other (gossip)	Nodes are not aware of each other



# Clustering is fairly young

## Cluster Manager

- Same complex code replicated over all nodes
- Failure modes:
  - Data rebalance breaks
  - Data corruption across all nodes
  - Improper balancing that cannot be fixed (easily)
  - Data authority failure




# Data Partitioning and Sharding Patterns

[Reference](#)



# Data partitioning

- **Horizontal partitioning** (often called *sharding*). In this strategy each partition is a data store in its own right, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers in an ecommerce application.
- **Vertical partitioning**. In this strategy each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use, such as placing the frequently accessed fields in one vertical partition and the less frequently accessed fields in another.
- **Functional partitioning**. In this strategy data is aggregated according to how it is used by each bounded context in the system. For example, an ecommerce system that implements separate business functions for invoicing and managing product inventory might store invoice data in one partition and product inventory data in another.

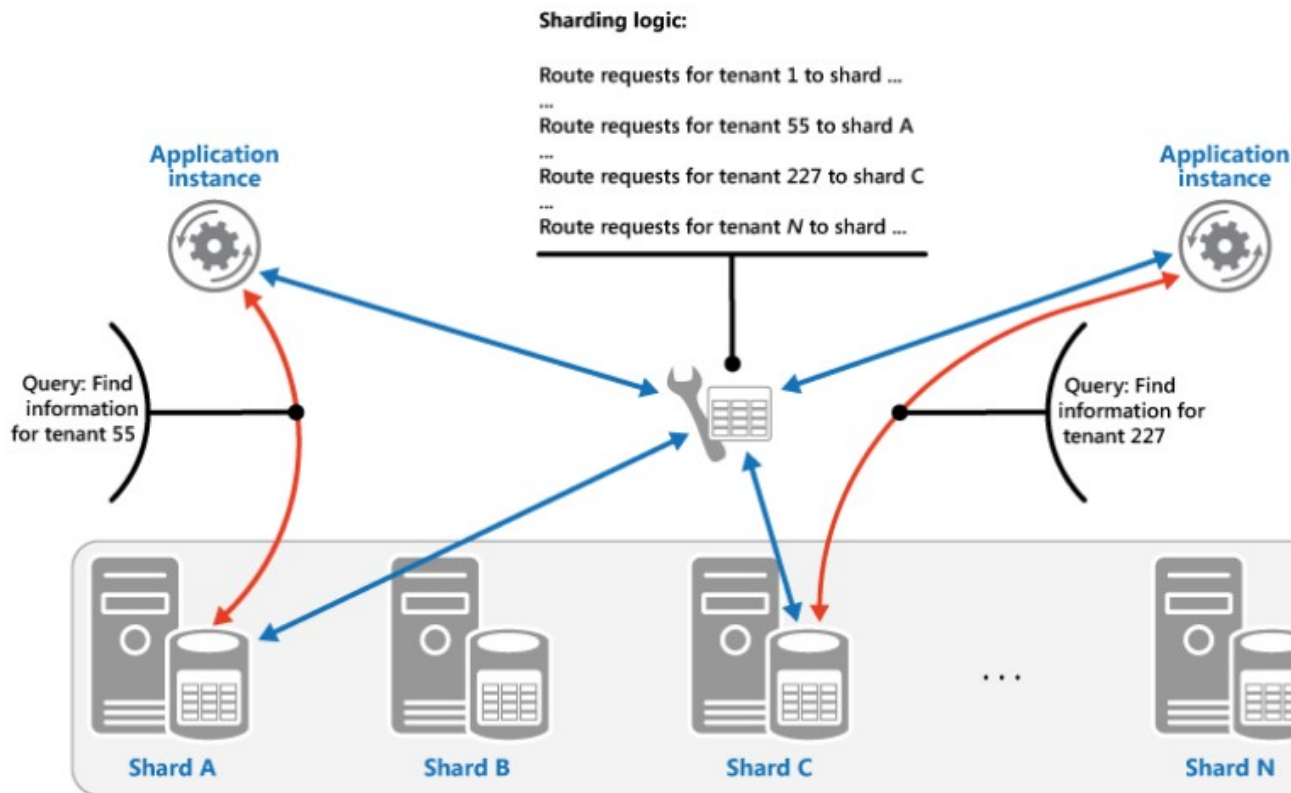


# Partition designing

1. [Designing Partitions for Scalability](#)
2. [Designing Partitions for Query Performance](#)
3. [Designing Partitions for Availability](#)

Click on each for more information

# Lookup strategy in sharding





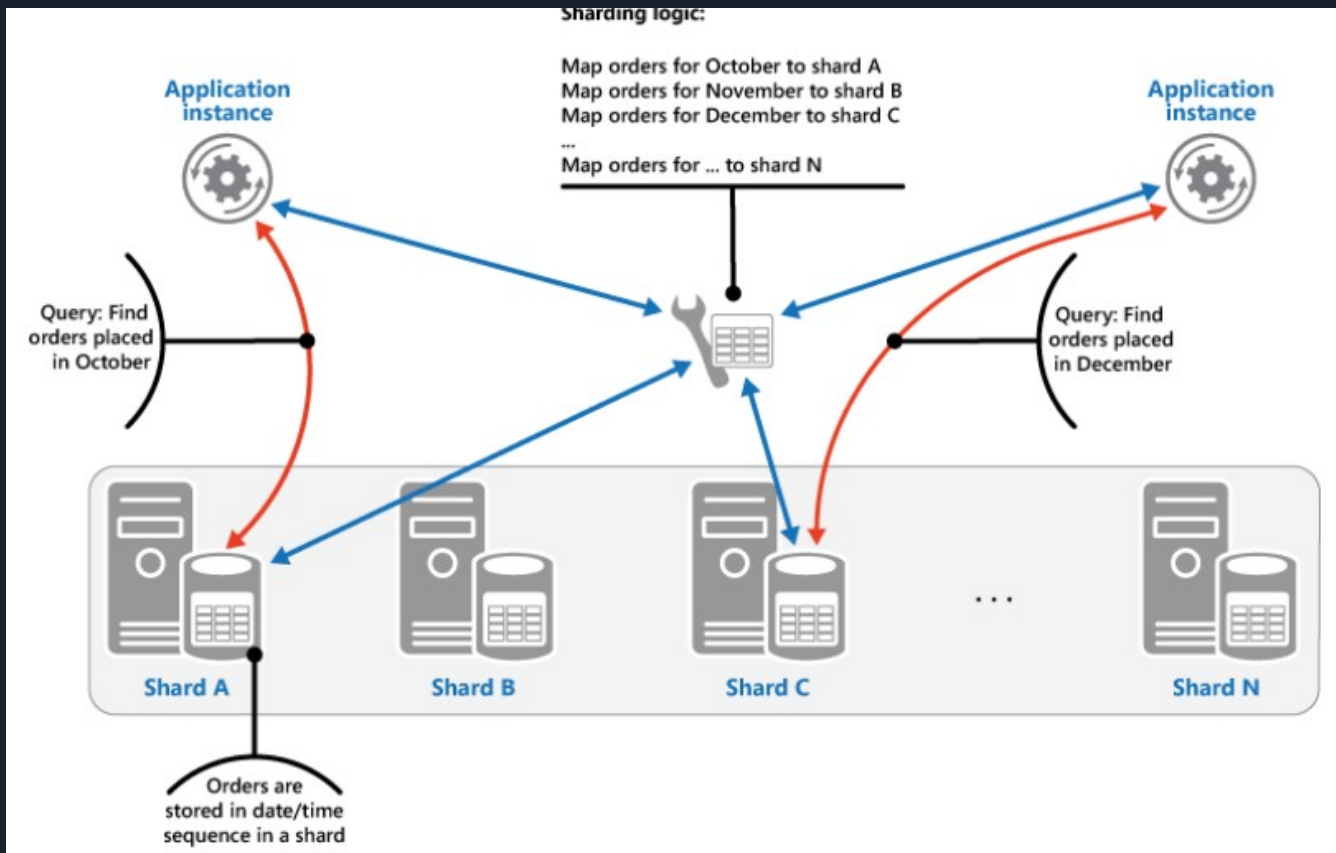


# Lookup strategy in sharding

In a multi-tenant application all the data for a tenant might be stored together in a shard by using the tenant ID as the shard key. Multiple tenants might share the same shard, but the data for a single tenant will not be spread across multiple shards.

The mapping between the shard key and the physical storage may be based on physical shards where each shard key maps to a physical partition. Alternatively, a technique that provides more flexibility when rebalancing shards is to use a virtual partitioning approach where shard keys map to the same number of virtual shards, which in turn map to fewer physical partitions.

# The range strategy



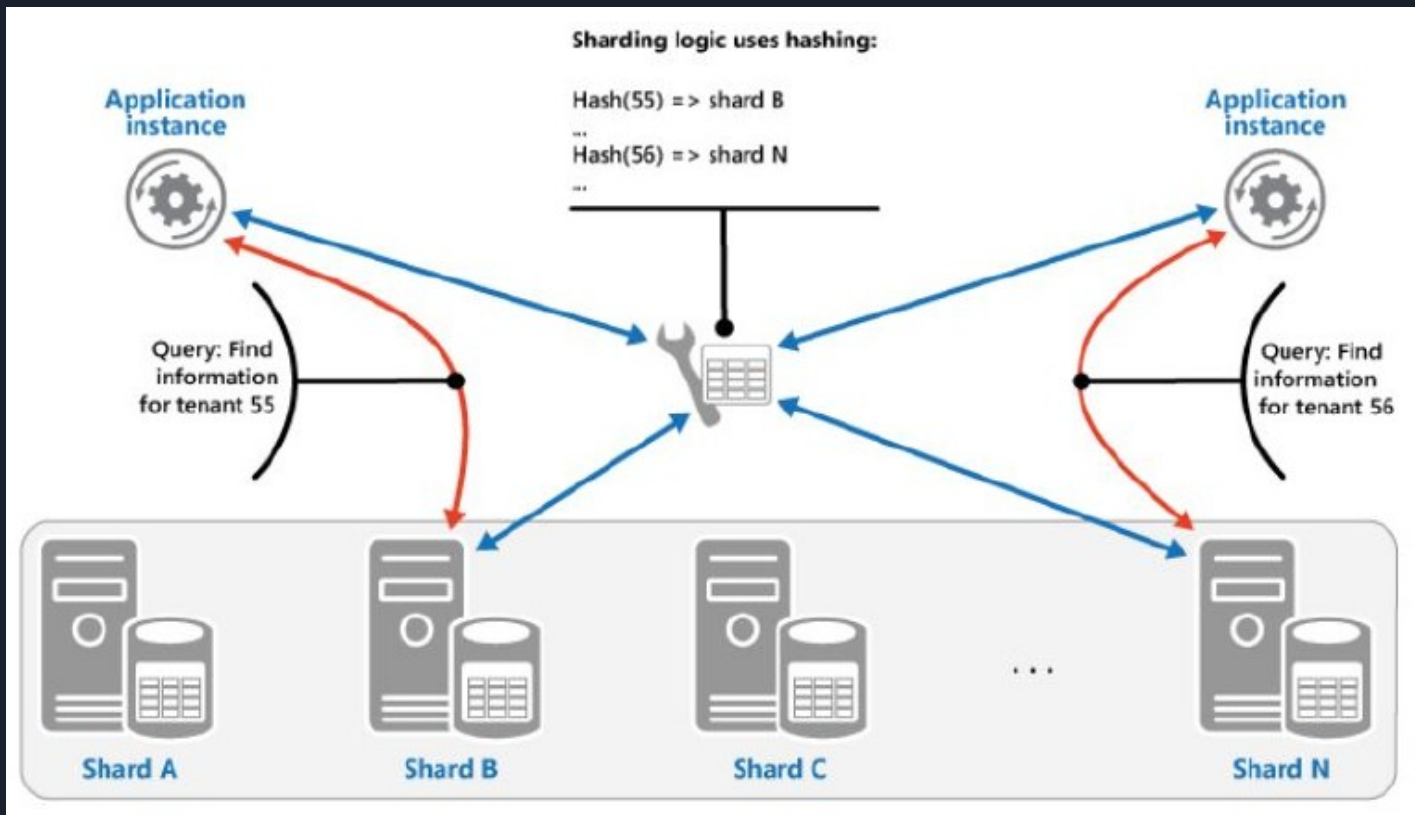


# The range strategy

It is useful for applications that frequently retrieve sets of items by using *range* queries (queries that return a set of data items for a shard key that falls within a given range). For example, if an application regularly needs to find all orders placed in a given month, this data can be retrieved more quickly if all orders for a month are stored in date and time order in the same shard.

The data for orders is naturally sorted when new orders are created and appended to a shard.

# Hash





# Hash

The purpose of this strategy is to reduce the chance of hotspots in the data. It aims to distribute the data across the shards in a way that achieves a balance between the size of each shard and the average load that each shard will encounter.



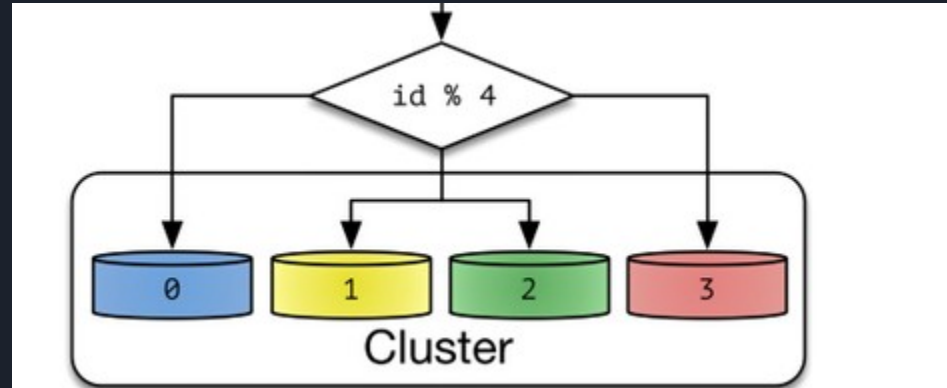
# Compare

Strategy	Advantages	Considerations
Lookup	<p>More control over the way that shards are configured and used.</p> <p>Using virtual shards reduces the impact when rebalancing data because new physical partitions can be added to even out the workload. The mapping between a virtual shard and the physical partitions that implement the shard can be modified without affecting application code that uses a shard key to store and retrieve data.</p>	<p>Looking up shard locations can impose an additional overhead.</p>
Range	<p>Easy to implement and works well with range queries because they can often fetch multiple data items from a single shard in a single operation.</p> <p>Easier data management. For example, if users in the same region are in the same shard, updates can be scheduled in each time zone based on the local load and demand pattern.</p>	<p>May not provide optimal balancing between shards.</p> <p>Rebalancing shards is difficult and may not resolve the problem of uneven load if the majority of activity is for adjacent shard keys.</p>
Hash	<p>Better chance of a more even data and load distribution.</p> <p>Request routing can be accomplished directly by using the hash function. There is no need to maintain a map.</p>	<p>Computing the hash may impose an additional overhead.</p> <p>Rebalancing shards is difficult.</p>

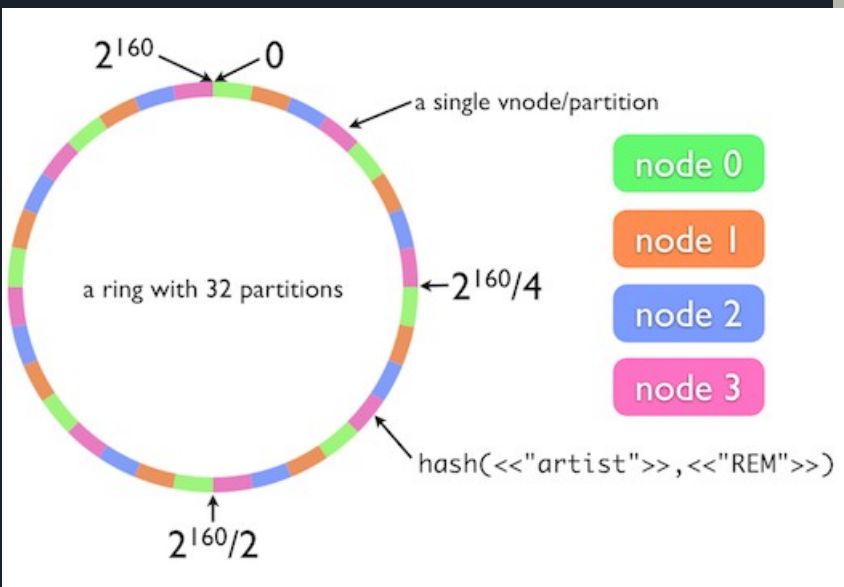
# Algorithmic sharding

Algorithmic sharding distributes data by its sharding function only. It doesn't consider the payload size or space utilization. To uniformly distribute data, each partition should be similarly sized. Fine grained partitions reduce hotspots—a single database will contain many partitions, and the sum of data between databases is statistically likely to be similar. For this reason, algorithmic sharding is suitable for key-value databases with homogeneous values.

Resharding data can be challenging. It requires updating the sharding function and moving data around the Cluster. Doing both at the same time while maintaining consistency and availability is hard.



# Consistent hashing



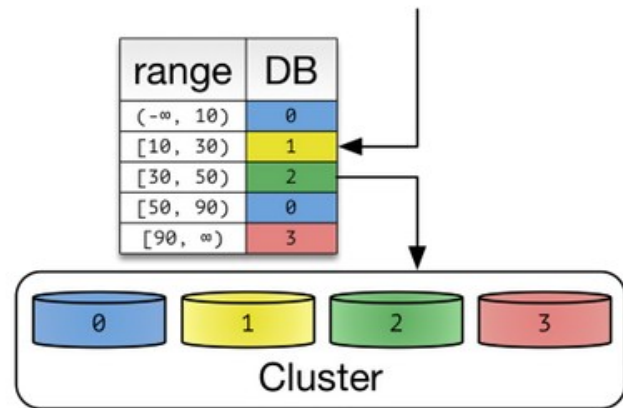


# Dynamic sharding

In dynamic sharding, an external **locator service** determines the location of entries.

Queries without a partition key will need to search all databases.

Dynamic sharding is more resilient to nonuniform distribution of data.



A dynamic sharding scheme using range based partitioning.

# Entity groups

The concept of entity groups is very simple. Store related entities in the same partition to provide additional capabilities within a single partition. Specifically:

1. Queries within a single physical shard are efficient.
2. Stronger consistency semantics can be achieved

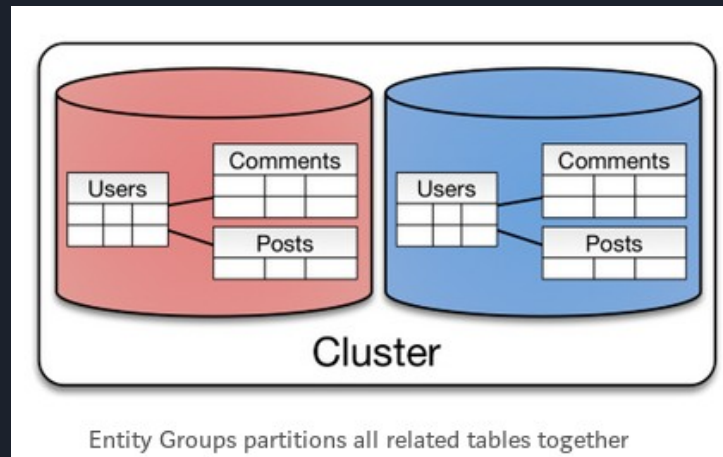
within a shard.


In a typical web application data is naturally isolated per user. Partitioning

by user gives scalability of sharding while retaining most of its flexibility.

cross-partition query may be required frequently and efficiently.

In this case, data needs to be stored in multiple partitions






# Sharding Considerations and Techniques

- 
- Single-Shard: Each database operation (read or write) is performed against a single "shard," such as for a single customer transaction.
  - Multi-Shard: This typically applies to analytic queries, performed in parallel across one or more "shards," enabling impressive performance results.

The above are scaling solutions increase database capacity exponentially. The cost of such database resources might not be affordable for small and medium-sized enterprises. The cost for upgrading from a 2-instance solution to a 4-instance solution doubles and it doubles again when upgrading from the 4-instance solution to an 8-instance solution.

Resharding data can be challenging. It requires updating the sharding function and moving data around the cluster. Doing both at the same time while maintaining consistency and availability is hard. Clever choice of sharding function can reduce the amount of transferred data. Consistent Hashing is such an algorithm.



Examples of such system include Memcached. Memcached is not sharded on its own, but expects client libraries to distribute data within a cluster. Such logic is fairly easy to implement at the application level.

## Application Changes Required for MySQL Sharding

Sharding usually requires significant application changes as well. Applications moving to a MySQL sharded array now have to handle the following:

- Application query routing to the correct shard.
- Cross-node transactions and ACID transactionality.

Part of sharding is creating and maintaining the mapping between sharding keys, data partitions, databases, and nodes. This really shouldn't be done in application code, because this mapping will change often, especially in a growing sharded array. Shard splits, shard migrations, instance replacement, and sharding key changes all will change this mapping. Ideally, this mapping should be done in a very fast lookup because potentially, this information is needed for each query. Thus, this information is often located in an in-memory NoSQL database such as Redis or Memcache.




## Not All Tables Are Sharded

Some tables are not sharded, but rather replicated into each shard. These tables contain *reference data*, which is not specific to any particular business entity and is *read-mostly*. For example, a list of zip codes is reference data; the application may use this data to determine the city from a zip code. We duplicate reference data in each shard to maintain shard autonomy: all of the data needed for queries must be in the shard.

The rest of the tables are typically sharded. Unlike reference data, any given row of a sharded table is stored on exactly one shard node; there is no duplication.

The sharded tables typically include those tables responsible for the bulk of the data size and database traffic. Reference data typically does not change very often and is much smaller than business data.




most recent year was modified heavily, which caused a lot of contention on that mutex. Subpartitioning by hash helped chop the data into smaller pieces and alleviated the problem.

Other partitioning techniques we've seen include:

- You can partition by key to help reduce contention on InnoDB mutexes.
- You can partition by range using a modulo function to create a round-robin table that retains only a desired amount of data. For example, you can partition date-based data by day modulo 7, or simply by day of week, if you want to retain only the most recent days of data.
- Suppose you have a table with an autoincrementing `id` primary key, but you want to partition the data temporally so the “hot” recent data is clustered together. You can't partition by a timestamp column unless you include it in the primary key, but that defeats the purpose of a primary key. You can partition by an expression such as `HASH(id DIV 1000000)`, which creates a new partition for each million rows inserted. This achieves the goal without requiring you to change the primary key. It has the added benefit that you don't need to constantly create partitions to hold new ranges of dates, as you'd need to do with range-based partitioning.






### *Scan the data, don't index it*

You can create tables without indexes and use partitioning as the only mechanism to navigate to the desired kind of rows. As long as you always use a **WHERE** clause that prunes the query to a small number of partitions, this can be good enough. You'll need to do the math and decide whether your query response times will be acceptable, of course. The assumption here is that you're not even trying to fit the data in memory; you assume that anything you query has to be read from disk, and that that data will be replaced soon by some other query, so caching is futile. This strategy is for when you have to access a lot of the table on a regular basis. A caveat: for reasons we'll explain a bit later, you usually need to limit yourself to a couple of hundred partitions at most.

### *Index the data, and segregate hot data*

If your data is mostly unused except for a “hot” portion, and you can partition so that the hot data is stored in a single partition that is small enough to fit in memory along with its indexes, you can add indexes and write queries to take advantage of them, just as you would with smaller tables.





The queries we've observed to suffer the worst from this type of overhead are row-by-row inserts. For every row you insert into a table that's partitioned by range, the server has to scan the list of partitions to select the destination. You can alleviate this problem by limiting how many partitions you define. In practice, a hundred or so works okay for most systems we've seen.

Other partition types, such as key and hash partitions, don't have the same limitation.

#### *Opening and locking partitions can be costly*

Opening and locking partitions when a query accesses a partitioned table is another type of per-partition overhead. Opening and locking occur before pruning, so this isn't a prunable overhead. This type of overhead is independent of the partitioning type and affects all types of statements. It adds an especially noticeable amount of overhead to short operations, such as single-row lookups by primary key. You can avoid high per-statement costs by performing operations in bulk, such as using multirow inserts or `LOAD DATA INFILE`, deleting ranges of rows instead of one at a time, and so on. And, of course, limit the number of partitions you define.



# ProxySQL and Sharding

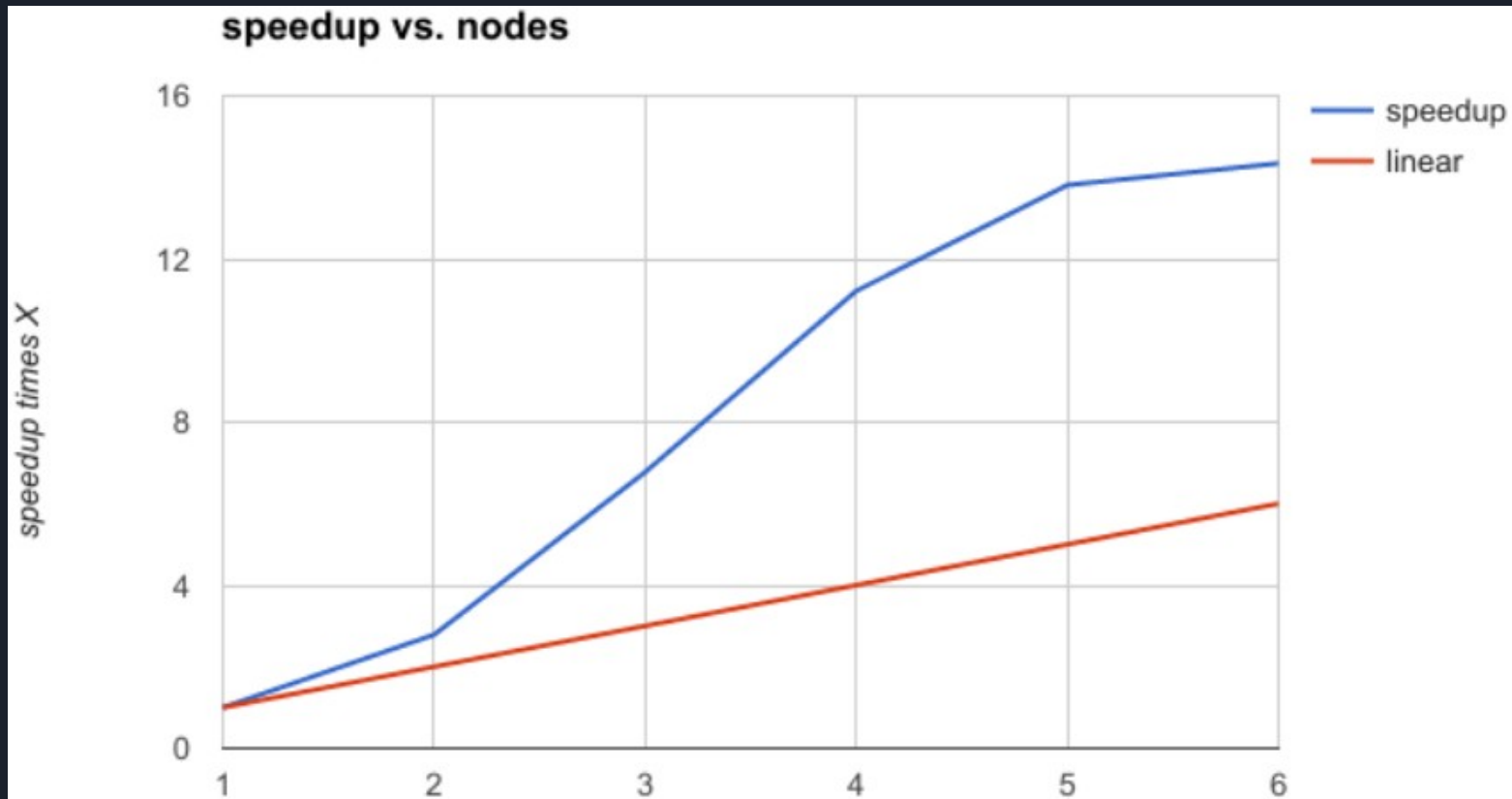



# Kind of scenarios

No matter what the requirements, the sharding exercise can be summarized in a few different categories.

- By splitting the data inside the same container (like having a shard by State where each State is a schema)
- By physical data location (this can have multiple MySQL servers in the same room, as well as having them geographically distributed)
- A combination of the two, where I do split by State using a dedicated server, and again split by schema/table by whatever (say by gender)

Done via query routing and host group mapping ( something like lookup service )






# ID Generation Approaches

[Refrence](#)




# Essential features to keep in mind

- Generated IDs should be sortable by time (so a list of photo IDs, for example, could be sorted without fetching more information about the photos)
- IDs should ideally be 64 bits (for smaller indexes, and better storage in systems like Redis)
- The system should introduce as few new 'moving parts' as possible—a large part of how we've been able to scale Instagram with very few engineers is by choosing simple, easy-to-understand solutions that we trust.
  - Simplicity in architecture and solution



# Generate IDs in web application

- This approach leaves ID generation entirely up to your application, and not up to the database at all.
- MongoDB's ObjectId, which is 12 bytes long and encodes the timestamp as the first component.
- Another popular approach is to use UUIDs.



# Generate IDs in web application

- Pros:
  - Each application thread generates IDs independently, minimizing points of failure and contention for ID generation
  - If you use a timestamp as the first component of the ID, the IDs remain time-sortable
- Cons:
  - Generally requires more storage space (96 bits or higher) to make reasonable uniqueness guarantees
  - Some UUID types are completely random and have no natural sort





# Generate IDs through dedicated service

- Twitter's Snowflake, a Thrift service that uses Apache ZooKeeper to coordinate nodes and then generates 64-bit unique IDs
  - <https://github.com/twitter-archive/snowflake/releases>
  - Keep in mind this might be a good implementation from Sony
    - <https://github.com/sony/sonyflake>



# Generate IDs through dedicated service

- Pros:
  - Snowflake IDs are 64-bits, half the size of a UUID
  - Can use time as first component and remain sortable
  - Distributed system that can survive nodes dying
- Cons:
  - Would introduce additional complexity and more 'moving parts' (ZooKeeper, Snowflake servers) into our architecture



# DB Ticket Servers

- Uses the database's auto-incrementing abilities to enforce uniqueness.
- Flickr uses this approach, but with two ticket DBs (one on odd numbers, the other on even) to avoid a single point of failure.



# DB Ticket Servers

- Pros:
  - DBs are well understood and have pretty predictable scaling factors
- Cons:
  - Can eventually become a write bottleneck (though Flickr reports that, even at huge scale, it's not an issue).
  - An additional couple of machines (or EC2 instances) to admin
  - If using a single DB, becomes single point of failure. If using multiple DBs, can no longer guarantee that they are sortable over time.




# Instagram approach

- Of all the approaches above, Twitter's Snowflake came the closest, but the additional complexity required to run an ID service was a point against it. Instead, we took a conceptually similar approach, but brought it inside PostgreSQL.



# Instagram DB sharding overview

- Our sharded system consists of several thousand ‘logical’ shards that are mapped in code to far fewer physical shards.
- Using this approach, we can start with just a few database servers, and eventually move to many more, simply by moving a set of logical shards from one database to another, without having to re-bucket any of our data.

- 
- We used Postgres' schemas feature to make this easy to script and administrate.
  - Table names must only be unique per-schema, not per-DB, and by default Postgres places everything in a schema named 'public'.
  - Each 'logical' shard is a Postgres schema in our system, and each sharded table (for example, likes on our photos) exists inside each schema.
    - We've delegated ID creation to each table inside each shard, by using PL/PGSQL, Postgres' internal programming language, and Postgres' existing auto-increment functionality.



# ID Generation Format

- Each of our IDs consists of:
- 
- 41 bits for time in milliseconds (gives us 41 years of IDs with a custom epoch)
- 13 bits that represent the logical shard ID
- 10 bits that represent an auto-incrementing sequence, modulus 1024. This means we can generate 1024 IDs, per shard, per millisecond






# Examples

- Let's walk through an example: let's say it's September 9th, 2011, at 5:00pm and our 'epoch' begins on January 1st, 2011. There have been 1387263000 milliseconds since the beginning of our epoch, so to start our ID, we fill the left-most 41 bits with this value with a left-shift:
  - $Id = 1387263000 \ll (64-41)$
- Next, we take the shard ID for this particular piece of data we're trying to insert. Let's say we're sharding by user ID, and there are 2000 logical shards; if our user ID is 31341, then the shard ID is  $31341 \% 2000 \rightarrow 1341$ . We fill the next 13 bits with this value:
  - $Id |= 1341 \ll (64-41-13)$
- Finally, we take whatever the next value of our auto-increment sequence (this sequence is unique to each table in each schema) and fill out the remaining bits. Let's say we'd generated 5,000 IDs for this table already; our next value is 5,001, which we take and mod by 1024 (so it fits in 10 bits) and include it too:
  - $Id |= (5001 \% 1024)$

We now have our ID, which we can return to the application server using the RETURNING keyword as part of the INSERT.



Here's the PL/PGSQL that accomplishes all this (for an example schema insta5):

```
CREATE OR REPLACE FUNCTION insta5.next_id(OUT result bigint) AS $$
DECLARE
    our_epoch bigint := 1314220021721;
    seq_id bigint;
    now_millis bigint;
    shard_id int := 5;
BEGIN
    SELECT nextval('insta5.table_id_seq') %% 1024 INTO seq_id;


    SELECT FLOOR(EXTRACT(EPOCH FROM clock_timestamp()) * 1000)
    INTO now_millis;
    result := (now_millis - our_epoch) << 23;
    result := result | (shard_id <<10);
    result := result | (seq_id);
END;
$$ LANGUAGE PLPGSQL;
```

And when creating the table, we do:

```
CREATE TABLE insta5.our_table (
    "id" bigint NOT NULL DEFAULT insta5.next_id(),
    ...rest of table schema...
)
```

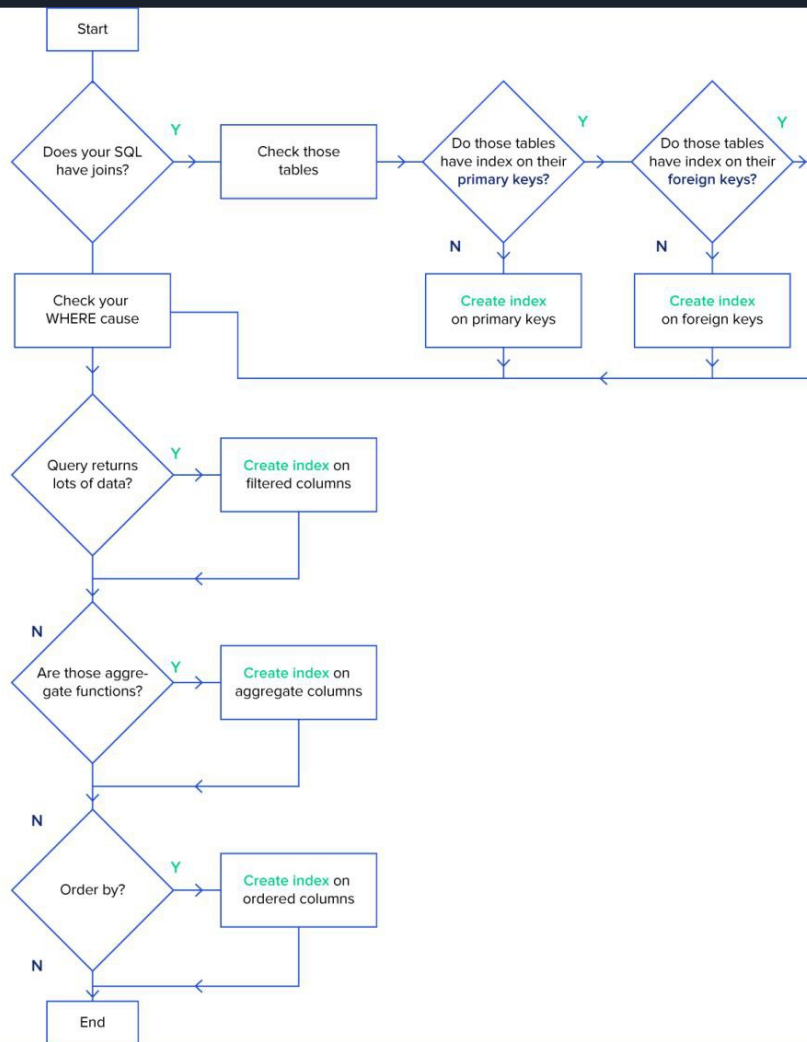



# Query Optimization



```
mysql> SELECT p.project_id, p.name, IFNULL(count(distinct r.relation_
      FROM (SELECT * FROM projects WHERE source='Apache') AS p
      LEFT JOIN relations AS r on (p.project_id=r.project_id)
      WHERE r.relation_type='INSIDE'
      GROUP BY p.project_id
      ...
      132 rows in set (57.26 sec)
```

According to the explanation, in the first query, the selection on the projects table is done first. Because it involves only a couple of hundred of thousands of rows, the resulting table can be kept in memory; the following join between the resulting table and the very large relations table on the indexed field is fast. We're all good. However, in the second query, the explanation tells us that, first, a selection is done on the relations table (effectively, relations WHERE relation\_type='INSIDE'); the result of that selection is huge (millions of rows), so it doesn't fit in memory, so MySQL uses a temporary table; the creation of that table takes a very long time... catastrophe!





Note that if your tables are constantly hammered by `INSERT` , `UPDATE` , and `DELETE` , you should be careful when indexing—you could end up [decreasing performance](#) as all indexes need to be modified after these operations.

Further, DBAs often drop their SQL indexes before performing batch inserts of million-plus rows to [speed up the insertion process](#). After the batch is inserted, they then recreate the indexes. Remember, however, that dropping indexes will affect every query running in that table; so this approach is only recommended when working with a single, large insertion.



## SQL Tuning: Avoid Coding Loops

Imagine a scenario in which 1000 queries hammer your database in sequence. Something like:

```
for (int i = 0; i < 1000; i++)  
{  
    SqlCommand cmd = new SqlCommand("INSERT INTO TBL (A,B,C) VALUES..  
    cmd.ExecuteNonQuery();  
}
```

You should [avoid such loops](#) in your code. For example, we could transform the above snippet by using a unique `INSERT` or `UPDATE` statement with multiple rows and values:




Here's an example of a correlated subquery:

```
SELECT c.Name,  
       c.City,  
       (SELECT CompanyName FROM Company WHERE ID = c.CompanyID) AS Cor  
FROM Customer c
```

In particular, the problem is that the inner query (`SELECT CompanyName...`) is run for *each* row returned by the outer query (`SELECT c.Name...`). But why go over the `Company` again and again for every row processed by the outer query?





A more efficient SQL performance tuning technique would be to refactor the correlated subquery as a join:

```
SELECT c.Name,  
       c.City,  
       co.CompanyName  
FROM Customer c  
      LEFT JOIN Company co  
            ON c.CompanyID = co.CompanyID
```


In this case, we go over the `Company` table just once, at the start, and JOIN it with the `Customer` table. From then on, we can select the values we need ( `co.CompanyName` ) more efficiently.



## SQL Tuning: Select Sparingly

One of my favorite SQL optimization tips is to avoid `SELECT *` !  
Instead, you should individually include the specific columns that you need. Again, this sounds simple, but I see this error all over the place.

When you have to join a large table and there are conditions on said table, you can increase database performance by transferring your data in a temp table, and then making a join on *that*. Your temp table will have fewer rows than the original (large) table, so the join will finish faster!




Imagine a customer table with millions of records. You have to make a join on a specific region. You can achieve this by using a

`SELECT INTO` statement and then joining with the temp table:

```
SELECT * INTO #Temp FROM Customer WHERE RegionID = 5  
SELECT r.RegionName, t.Name FROM Region r JOIN #Temp t ON t.RegionID =
```

*(Note: some SQL developers also avoid using `SELECT INTO` to create temp tables, saying that this command locks the tempdb database, disallowing other users from creating temp tables. Fortunately, this is [fixed in 7.0 and later.](#))*



As an alternative to temp tables, you might consider using a subquery as a table:

```
SELECT r.RegionName, t.Name FROM Region r
JOIN (SELECT * FROM Customer WHERE RegionID = 5) AS t
ON t.RegionID = r.RegionID
```

But wait! There's a problem with this second query. As described above, we should only be including the columns we need in our subquery (i.e., not using `SELECT *`). Taking that into account:

```
SELECT r.RegionName, t.Name FROM Region r
JOIN (SELECT Name, RegionID FROM Customer WHERE RegionID = 5) AS t
ON t.RegionID = r.RegionID
```



## SQL Tuning: “Does My Record Exist?”

This SQL optimization technique concerns the use of `EXISTS()`. If you want to check if a record exists, use `EXISTS()` instead of `COUNT()`. While `COUNT()` scans the entire table, counting up all entries matching your condition, `EXISTS()` will exit as soon as it sees the result it needs. This will give you [better performance and clearer code](#).



It depends on the operations that occur on the table.

If there's lots of SELECTs and very few changes, index all you like.... these will (potentially) speed the SELECT statements up.

If the table is heavily hit by UPDATES, INSERTs + DELETES ... these will be very slow with lots of indexes since they all need to be modified each time one of these operations takes place

Having said that, you can clearly add a lot of pointless indexes to a table that won't do anything. Adding B-Tree indexes to a column with 2 distinct values will be pointless since it doesn't add anything in terms of looking the data up. The more unique the values in a column, the more it will benefit from an index.

I usually proceed like this.

1. Get a log of the *real* queries run on the data on a typical day.
2. Add indexes so the most important queries hit the indexes in their execution plan.
3. Try to avoid indexing fields that have a lot of updates or inserts
4. After a few indexes, get a new log and repeat.

As with all any optimization, I stop when the requested performance is reached (this obviously implies that point 0. would be getting specific performance requirements).



### 3. Select More Fields to Avoid SELECT DISTINCT

**SELECT DISTINCT** is a handy way to remove duplicates from a query. **SELECT DISTINCT** works by **GROUP**ing all fields in the query to create distinct results. To accomplish this goal however, a large amount of processing power is required. Additionally, data may be grouped to the point of being inaccurate. To avoid using **SELECT DISTINCT**, more fields should be selected to create unique results.




## 4. Create Joins with INNER JOIN Rather than WHERE

Some SQL developers prefer to make joins with **WHERE** clauses, such as the following:

```
SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate  
FROM Customers, Sales  
WHERE Customers.CustomerID = Sales.CustomerID
```






Some DBMS systems are able to recognize **WHERE** joins and automatically run them as **INNER JOINS** instead. In those DBMS systems, there will be no difference in performance between a **WHERE** join and **INNER JOIN**. However, **INNER JOIN** is recognized by all DBMS systems. Your DBA will advise you as to which is best in your environment.


## 8. Run Analytical Queries During Off-Peak Times

In order to minimize query impact on the production database, talk to a DBA about scheduling the query to run at an off-peak time. The query should run when concurrent users are at their lowest number, which is typically the middle of the night (3 – 5 AM).

- 
- To retrieve rows from other tables when performing joins. MySQL can use indexes on columns more efficiently if they are declared as the same type and size. In this context, VARCHAR and CHAR are considered the same if they are declared as the same size. For example, **VARCHAR(10)** and **CHAR(10)** are the same size, but **VARCHAR(10)** and **CHAR(15)** are not.

the smaller integer types if possible to get smaller tables. MEDIUMINT is often a better choice than INT because a MEDIUMINT column uses 25% less space.

- Declare columns to be **NOT NULL** if possible. It makes SQL operations faster, by enabling better use of indexes and eliminating overhead for testing whether each value is **NULL**. You also save some storage space, one bit per column. If you really need **NULL** values in your tables, use them. Just avoid the default setting that allows **NULL** values in every column.

- 
1. Create a primary key as a clustered index for every table. Fortunately, Enterprise Manager clusters primary keys by default. You should also set up an index for any column that is a foreign key.
  2. When you reference objects in T-SQL, make it a habit to always owner qualify them. For example, you would use `dbo.sysdatabases` as opposed to just `sysdatabases`.
  3. Use `SET NOCOUNT ON` at the top of every procedure and `SET NOCOUNT OFF` at the bottom of every procedure.
  4. Unless you're writing banking software, then don't bother with locking. Using `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED` at the top and reverting to `READ COMMITTED` at the bottom of your stored procedures is more effective than using the `NOLOCK` hint.
  5. Feel free to use transactions when needed, but don't allow any user interaction while they are in progress. It's best to keep all of your transactions inside a stored procedure.
  6. Don't use temp tables unless you absolutely need them.
  7. Scan your code for uses of the `NOT IN` command and replace them with a left outer join.
  8. Speaking of which, make it a habit to review your code before and after each change you make.
  9. Find ways to lower the amount of required round trips to the server. For example, try returning multiple `ResultSets`.
  10. Don't bother using index hints or join hints.
  11. Instead of using `SELECT *` statements, you should just individually specify each column you require even if you need every column in a table.
  12. When checking to see if a record exists, use `EXISTS()` instead of `COUNT()`. They are both effective, but `EXISTS()` stops running automatically once it finds the requested record, which will result in better performance and cleaner code.



# Thanks for your attention!

Feel free to ask any questions  
[navidmalekedu@gmail.com](mailto:navidmalekedu@gmail.com)