Deep Database Dive

Author: Shaun Thomas
Date: November 28th, 2016
Venue: Database Design and Management

Your Presenter

Shaun M. Thomas
Database Architect
Peak6 Investments

I apologize in advance.

Who is This Guy?!

- DBA since 2001
- Presented at Postgres Open 2011-2016
- Postgres 9 High Availability Cookbook
- PG Phriday (http://planet.postgresql.org)

While I tend to work exclusively with PostgreSQL, database modeling, scaling, and high availabilty considerations are universal concepts. In many ways, MySQL is just another database.

In The Field

Biggest problems we face:

- Workload
- Accumulation
- Security
- Distribution

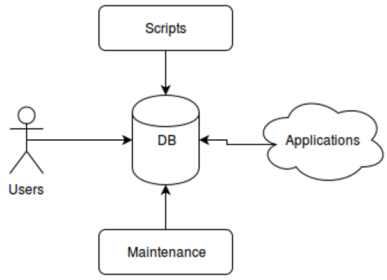
Look familiar?

WASD

WASD isn’t just for playing games! To be fair, I stretched to make this acronym; it’s not actually used anywhere in industry. There’s a first time for everything, though!

Workload

Usually looks like this



This really is just a subset of all access vectors. A good design will have to account for all of these.

Accumulation

This is our data.

Just throw it on the pile!

This isn't much of a stretch from the truth. I've seen databases exceed 50TB because they're just gathering data from multiple sources for all eternity. The trick is making the pile usable.

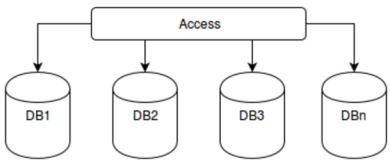
Security

- Hackers
- Encryption
- Privileges
- History

Good designs account for security, too. What data should be accessible, and to who? What grants should be baked into system? Should we include stored procedures to capture DML activity into auditing logs? Do we need double-bookkeeping? How much encryption do we need? All questions that need answers, even if the answers is "don't worry about it."

Distribution

Where to put our pile?



Do we care if everything is in one gargantuan heap, or should we spend some time adapting the design so it can account for distribution or sharding techniques?

Let’s Make Something!

Oh boy…

A voting system is topical, and we all know how it works. Hopefully.

How are Elections Done?

Things we *need*

- Elections
- Candidates
- Voters
- Votes

These are the essential elements of an election. We can not proceed without them, even in a fake, simulated system.

Election Extras

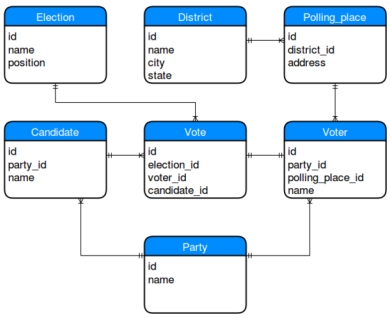
Nice data to have

- Districts
- Polling places
- Parties

We can include a few extra data elements that help describe the election and its components. Again, this isn't a real system, and we're operating at a fairly high level. There's usually a lot more involved than this.

Basic Diagram

Let's start here



This is the initial diagram. Voters can be associated with a party. Voters are assigned a polling place, and that polling place exists within a specific voting district. Elections do not represent every item on the ballot, just each individual item. We're not accounting for ballots in our voting engine. We can simulate ballots by holding multiple simultaneous elections with individual names.

Voters may spend their vote on a specific candidate / initiative for each election. Remember, we're keeping it simple.

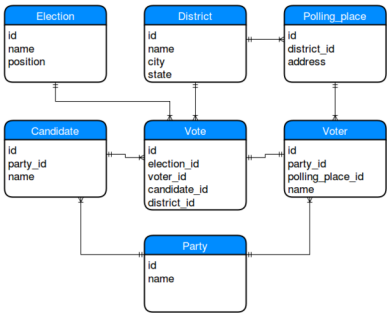
Early Problem

- Find the district of a vote.
- Three joins!
- Let's denormalize a bit.

We need to get the district for a particular vote to represent data by region, but that takes three joins. Eww. Let's fix that, at least.

Iterative Design

That's better!



Notice anything else? Technically we should have a "person" table, because a candidate can also be a voter. However, our candidate can really be a ballot initiative, a local tax, or voter-supplied write-in, or anything else. In that case, our model is actually more correct, though perhaps in that case, we should ensure party_id is optional, as not all elections have a party association.

This doesn't apply to candidates though, as unaffiliated candidates would list "Independent" as their party.

A Challenger Appears

Remember WASD!

- W: What are the access vectors?
- A: How much data are we gathering?
- S: How do we lock it down?
- D: Where do we put all of this data?

Let’s answer these questions.

WASD is our friend, here. The scalability of the design must account for responsiveness, or we might as well be using paper ballots.

Workload: Access Vectors

These are our actors:

- Voters (application)
- Results (application)
- Auditing (users)
- ETL (scripts)
- Backups (maintenance)

There’s a lot going on, and that means the database will be busy wherever it is. The model, permissions, indexes, and hardware need to account for all of these, and perhaps more.

Accumulation: Data Volume

What’s our scale?

- Thousands of cities
- Thousands of counties
- Millions of voters
- Dozens of initiatives (elections)

Data will build up quickly! Just one presidental election will represent over 120M rows entering our system in one single day. If even 10 other items sit on a single ballot, it’s really over one billion.

Security: Hack n’ Slash

Strictly control access!

- Read-only wherever possible
- WORM model (no update or delete)
- Double-bookkeeping
- Encrypted connections
- Encrypted storage
- Etc.

Lots of security concepts to consider. The database should effectively only allow writes. No updates or deletes anywhere. Further, only certain access vectors should be able to write at all. Everything should be encrypted to prevent vote buying. All writes to the database should go to two independent systems for auditing purposes. The connections between the application and the database, and the database and replicas should all be encrypted.

This is just on the database side! There's likely a VPN involved and other infrastructure-based tricks to lock down everything. Gotta keep out malicious users!

Distribution

Responsiveness matters!

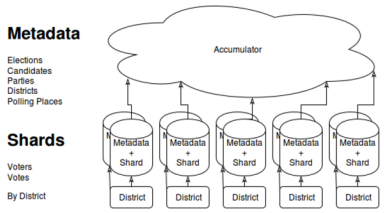
- Semi real-time results
- Too big for one system
- Metadata is relatively small
- Votes should be sharded

Can we really build and design a system to handle millions of users over thousands of cites and polling places and see results as they arrive? Multiply that by dozens of items on the ballot, and we have a slight scaling problem if we try and dump all of this into a single server.

Candidates, districts, polling places, the elections themselves, are all metadata that will total less than a million rows even over the course of several years. It's the voters and their votes that are our main concern; those need relocation.

The Real Model

Architecture isn't just for buildings



The model we've built will be implemented at a local level. Consider that voters and votes are represented by their local district. If we have a single database at each district, local results are handled immediately. Regional and national elections can invoke queries from downstream systems and present cumulative results. This makes our system naturally sharded, as each "node" can operate independently, but it's possible to combine data without much effort.

How Does it Work?

- Polling places collect data twice
- District systems act as aggregators
- Top-level accumulator can only read
- Audit systems only readable locally
- Massively parallel operation
- Districts are proportional

There's a lot more going on than in this diagram, of course. The point is that the metadata tables exists on every system, while votes and voters are isolated by district.

How Do We Use It?

1. Shard by shard
2. The accumulator
3. ETL-maintained fact tables

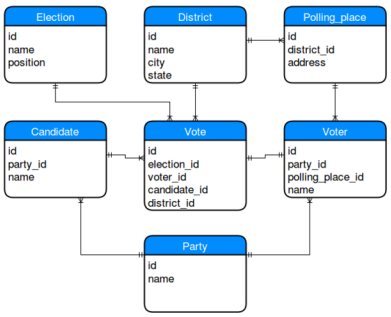
As users of these systems, we would have three approaches to use this data:

1. Obtain totals from a single shard. Basically each district has direct access to its own voting data.
2. Instruct the accumulator to aggregate from all shards at once. Or state-level accumulators can target their subset of district data.
3. Browse fact tables maintained via ETL during the election.

This includes the audit data, of course.

About Those Shards…

Where to add indexes?



There's the primary diagram again. Take a good look.

Index Selection

What makes a good indexable column?

- High cardinality
- Frequently used predicates
- Foreign identifiers (implicit predicates)

Indexes depend on high cardinality data and careful predicate analysis. The primary key (id) fields are obvious, but what else? Foreign keys are a good start for JOINs because they essentially add further predicates.

Cult of Cardinality

The basics:

- The amount of elements in a set
- More mean fewer rows match
- Less rows = less processing
- Less processing = faster results

What his basically boils down to, is that we want to index fields that have a lot of distinct values. Ideally, this would mean a 1-1 ratio between row count and distinct values. This is why primary keys provide the best performance. The more canddiates we can eliminate early, the fewer rows the database needs to aggregate or return.

JOIN-tastic

Consider this JOIN:

```
SELECT d.city, count(*)
  FROM district d
  JOIN vote v ON (v.district_id = d.id)
 WHERE d.state = 'IL'
 GROUP BY d.city;
```

It's really this to the query planner:

```
SELECT d.city, count(*)
  FROM district d, vote v
 WHERE d.state = 'IL'
   AND v.district_id = d.id
 GROUP BY d.city;
```

This is what we mean by implicit predicates. We build indexes based on high frequency predicates, and table joins are very relevant to that decision process. From the implied WHERE clause, we can see that there should be an index on the *district_id* column in the *vote* table.

Back to the Indexes

This would be our starting list:

- election.name
- candidate.name

- district.city, district.state
- party.name
- voter.name
- polling_place.district_id
- vote.*

We really do need all separate indexes on the columns in the vote table. Approach vector matters. We could justify all of these scenarios:

- Use the voter's ID code to quickly verify their vote was cast.
- Look up all information for a particular election.
- Gather votes for all elections a particular candidate has participated in.
- Find votes by city or state by district.

Each of these focuses on a different result set with distinct cardinality and thus separate indexing needs.

Where Am I?

This query needs no extra indexes:

```
SELECT p.address, d.city, d.state
  FROM voter v
  JOIN polling_place p ON (v.polling_place_id = p.id)
  JOIN district d ON (p.district_id = d.id)
 WHERE v.name = 'AXGY-2349-OODY-4271';
```

But this needs one on *district_id*:

```
SELECT p.address, d.city, d.state
  FROM district d
  JOIN polling_place p ON (p.district_id = d.id)
 WHERE d.state = 'IL';
```

Some JOINs are only descriptive. We already have the IDs from the original row set, we're just fetching from the referenced table to translate them. On the other hand, doing the lookup "backwards", means we have a list of primary district IDs and want a list of all polling places that use those values. See the difference?

The Bigger Picture

Remember the accumulator?

- Not your problem!
- Some application-level voodoo
- Basically does what we do, all at once
- Probably only cares about vote.election_id

The job of the accumulator is to aggregate from all shards, or some subset based on region. It's for global inquiries and its implementation details don't concern us. Hooray! But we have to know it exists in order to choose the best index candidates with the knowledge it will be likely sending a lot of generic aggregate queries that aren't very selective.

Say news crews use it for real-time national results. That usually means *election_id* is the only WHERE clause that'll come down the pipeline.

Reports by Design

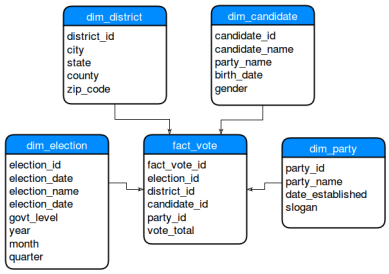
Global ad-hoc is bad!

- We need pre-aggregated summaries
- They should be maintained
- That means Extract Transform Load scripts
- Make the accumulator faster!

Knowing the accumulator is around means we need to precalculate some of the aggregates it fetches most often. Otherwise we'll be reading high percentages of our local tables as often as users invoke the accumulator. That's abusive and slow.

Search the Stars

Here's a simple star schema:



This is a basic set of fact tables commonly referred to as a star schema.

What's a Star Schema

Glad you asked!

- Several dimensions
- Extremely denormalized
- One aggregate "Fact" table at the center
- Orders of magnitude fewer rows

A star schema is generally a single aggregate table of "facts" surrounded by several dimensions. Each dimension fully describes the subject matter to a degree possibly greater than the original data. That information is then referenced by the fact table by the primary IDs with one or more cumulative totals that are the actual data points we're interested in.

As a consequence of this pre-aggregated design, instead of millions of rows per election, we may have a few thousand depending on how many districts are represented. If the accumulator has access to this, it doesn't need to query all of the shards all the time. In fact with the right design and implementation, we might not need the accumulator at all.

Finding Facts

Let's get the results for all elections in 2016 by state:

```
SELECT e.election_name, c.candidate_name, d.state,
       sum(v.vote_total) AS votes
  FROM fact_vote v
  JOIN dim_election e USING (vote_id)
  JOIN dim_candidate c USING (candidate_id)
  JOIN dim_district d USING (district_id)
 WHERE e.year = 2016
 GROUP BY e.election_name, c.candidate_name, d.state
 ORDER BY e.election_name, votes DESC;
```

Yes, each fact table column is indexed.

While fact tables are already aggregated, they're aggregated at the granularity of our chosen dimensions. We still need to summarize them if we leave any dimensions out. Otherwise, we know literally nothing about the fact table, as it's not even really a table, but a collection of facts related to the dimensions themselves. So our WHERE clauses target the dimensions and the JOIN takes care of the rest.

Still Too Slow!

And yet, fact tables still need further aggregation

Imagine if we didn't have an election, but market tick data. Each tick represents tens of thousands of transactions, and there are tens of thousands of ticks per day. A single year represents multiple TB of data. Even reduced to fact tables, access can be pretty slow.

There are further tricks up our sleeves.

Material Grill

Materialized views to the rescue.

- Static star-schema snapshot
- Completely flattens all common columns
- Allows further indexing
- Regular refreshes required

If we take all rows from all dimensions joined against the fact table, that might be one way to consider the data. That's a completely "flat" and clearly denormalized view of the data. It's literally the fastest way to get to the data, especially if common columns are indexed. There are no lookups beyond the initial match and filter steps, just results.

Matviews in Practice

MySQL doesn't support Mat views. Instead:

```
CREATE TABLE mv_election_flat AS
SELECT e.year, e.election_name, c.candidate_name, d.state,
       sum(v.vote_total) AS votes
  FROM fact_vote v
  JOIN dim_election e USING (vote_id)
  JOIN dim_candidate c USING (candidate_id)
  JOIN dim_district d USING (district_id)
 GROUP BY e.year, e.election_name, c.candidate_name, d.state
 ORDER BY e.year e.election_name, votes DESC;

CREATE INDEX idx_flat_election_name
    ON mv_election_flat (election_name);

CREATE INDEX idx_flat_state
    ON mv_election_flat (state);
```

Now we can get our results per state from this view, which is much smaller than even the fact table it's based on. Nice, eh?

More Information

- Cardinality
- Extract, Transform, Load
- Fact Tables

Questions