CPSC 324 Big Data Analytics
Final Project

Google Napa & Microsoft Moneyball

Colin McClelland

Napa - Overview

- “Analytical Data Management System” built by Google to enable high performance queries and scalability
- Built to improve Data Warehousing at Google
- Built to replace existing Mesa and take on its existing data and clients (along with some new clients)
  - A broader mandate
- Powers both internal and user facing applications

Napa - High Level Requirements

- Napa must be able to support:
  - Sub-second query performance
  - Petabytes of data
  - Continuous updates
  - Consistent query latency
  - Durability in the face of inevitable failure (fault tolerance)
- Napa had less flexibility in terms of design because it had to meet the requirements of existing applications (not a green field system)

Napa - High Level Design Principles

- Authors narrow the scope down to two (three) “bedrock” design principles that were derived from client needs

1. Robust Query Performance
   - Low latency AND low variance in latency regardless of load
2. Flexibility
   - Clients have different priorities (and budgets)
3. High throughput with respect to data ingestion
   - On the order of trillions of rows

Napa - Architecture + Fault Tolerance

- Built with existing Google infrastructure
  - Colossus: a table in Napa is a collection of Colossus files
  - Spanner: Metadata management and system state
  - F1 Query: query serving + large scale data operations
- Client databases are replicated across many datacenters
- Built to withstand failure at the data center levels (a guarantee)
- Clients access the optimal datacenter
- Systems in place to guarantee consistency

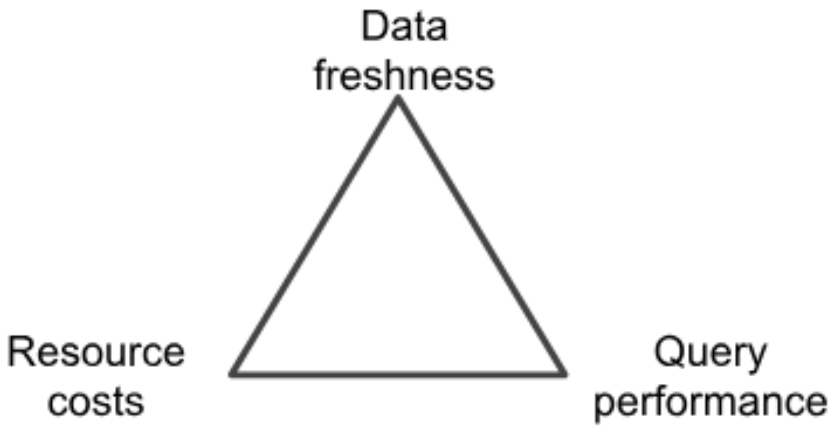
Napa - Terms

- Delta: incoming updates to a table
- Compaction: process on combining many small deltas into one large delta
- LSM tree: base tables and their views are maintained as a forests
  - Tables are collections of updates (deltas)

Napa - Three Frameworks

- Ingestion is responsible for committing updates (called deltas) into table
  - Allow ETL pipelines to insert data without any addl. overhead
- Storage incrementally applies the updates (deltas) to the corresponding tables/views
- Query Serving: answers client queries
  - Merges deltas at query time
  - Query latency is dependent on delta merging

Napa - Client Tradeoffs



- Napa offers clients a three-way tradeoff
- The flexibility tenant of Napa
- Query Performance: measured by latency
- Data Freshness: measured by the time delay between a row being added and it being available for querying
- Cost: computational resources used
- Clients can fine tune according to their priorities

Napa - Materialized Views

- The crux of Napa/its defining characteristic
- Napa maintains extensive materialized views for its base tables
- Uses these views to answer queries whenever possible as only references the base table as needed (much faster)
- Move coverage = faster query performance
  - But comes at the cost of additional overhead
- Updated consistently as new data is ingested
- A departure from other systems that have focused on efficient table scans

Napa - Compaction

- The process of combining many small deltas into one (merge sort)
- Large fan in, up to 1000 inputs before merge slows down
- Runs in the background
- Improves query performance by limiting query time merges, and reduces storage consumption
- However, compacting is computationally expensive
- The extent to which compacting occurs is dependent on client specification
- The low variation in query results is achieved by limiting the number of query time merges

Napa - Queryable Timestamp(QT)

- Denotes the point in time up to which data is available for querying
- Indicates how fresh the data is, most client require freshness on the order of 10s of minutes
- Also an indicator of system performance
  - Delay = Current time - QT
  - User and system react accordingly
- Only advances when system can ensure data meets performance requirements
- Clients don’t see any data after the QT
- Table level, database level, datacenter level

Napa - Maintaining View at Scale

- Since Napa heavily relies on materialized view, it must ensure that they are up to date and reflect changes made to the base table
- The responsibility of the storage subsystem
  - Along with compacting deltas and various fault tolerance + consistency checks
  - Must avoid data skew - fairly complicated process
- Exploits properties of the input data (namely ordering)
- Automatically adjusts to client budgets

Napa – Client Needs Inform System Behavior

- Paper describes three use cases for existing Napa clients

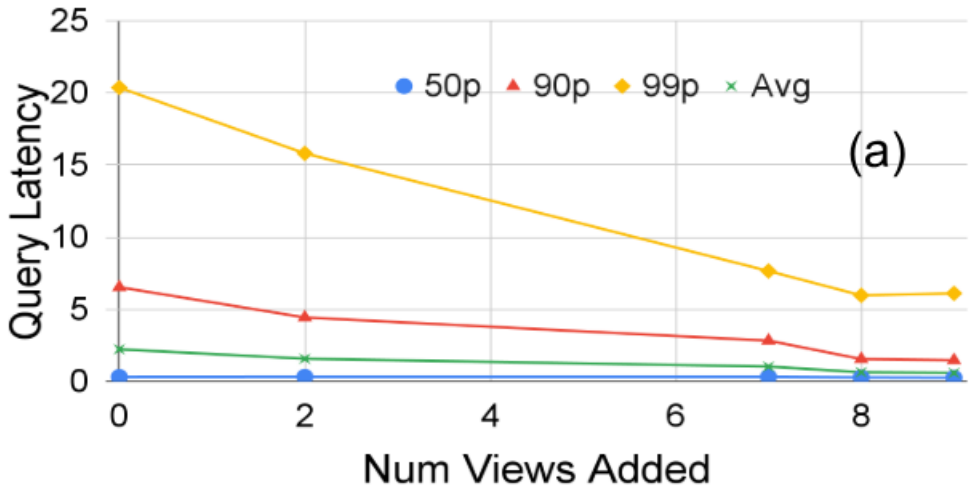
1. Client needs good (low latency) query performance at a moderate cost → have to sacrifice data freshness
   - Moderate view coverage & fewer deltas merged at query
2. Client needs fresh data but has lower performance standards
   - Less view coverage, but more deltas can be merged
3. Client needs good performance and fresh data → more computationally expensive
   - High view coverage & less query time merges

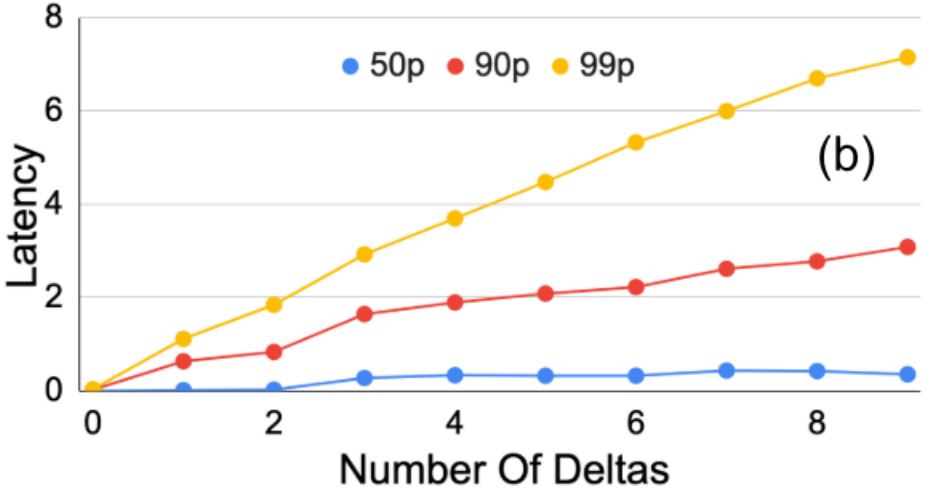
Napa - Optimizations

- Always use views over going to base table
- Push down filters and selects
  - Storage and Query engine workers not guaranteed to be in the same datacenter
- Inputs queries are partitioned into thousands of subqueries and run in parallel
- Limit I/O ops by detecting overlapping reads and only processing once
- Prefetching
  - Offline: newly ingested data cached for frequently queried tables
  - Online: Shadow query executor runs ahead of query executor
- Size based disk layout
- Emphasis is on tolerating stragglers rather than avoidance
  - Hedging and expected progress reports

Napa - System Performance & Takeaways

- Contributions: Materialized views, tunable params, QT
- Accommodates wider range of uses cases than Mesa, etc
  - Also provides lower latency and a lower cost to run
- So far Napa has been successful for all client workloads





Napa - Relation to Big Data Analytics

- Demonstrates the pillars (scalability, replication/recovery, coordination, consistency, placement)
- Further advances the idea of disaggregation
- Along with some newer ones: system built entirely out of existing components
- Data Warehousing still very much in play

Moneyball - Problem Setup

- Developed to improve user experience for Azure SQL serverless
- Serverless compute currently auto-scales to changes in workloads reactively
- When a database becomes idle its resources are reclaimed (paused) and when activity returns resources are returned (resumed)
- Due to the reactive approach resources may be unavailable for a time when the client returns
- There is also the issue of short pauses which are bad for a few reasons
  - Client may return soon thereafter and run into the above problem
  - Pause & Resume requires additional overhead costs

Moneyball - Solution

- Moneyball system has two main goals:

1. Proactively pause & resume resources to reduce latency by predicting pause/resume patterns using historical data
2. Avoid short pauses that would be ineffective and problematic for the client

- These goals must be met at a reasonable cost such that there is a net decrease in the total resources used

Moneyball - Challenges

- Many factors to consider when predicting pause/resume patterns
  - Length of historical data
- Two main goal are opposing
  - Incorrect resumes will increase the number of pauses, some of which will be short, but if pauses do not occur for long idle periods then resources will be wasted
- Many different usage pattens
- Moneyball is a middle ground approach
  - Between proactive resume & avoiding short pauses
  - Between COGS & QoS

Moneyball - Serverless Databases

- Serverless databases grouped into categories
  1. Stable: DB is paused or resumed a given percent of the time in the historical data (threshold)
  2. Patterned: a given percentage of DB's resumes and pauses occur within a specific window in the historical data
  3. Predictable: Stable or patterned
- Most serverless DBs are predictable based on a sample of several thousand
  - 74% stable > 90% of the time
  - 3% follow pattern > 90% of time
  - 23% remaining are unpredictable

Moneyball - Proactive Resume & Pause

- Two algorithms considered to proactively resume a DB for a given window (e.g. Wednesdays from 5:40 am - 9:20 am)

1. Probability : $\frac{\text{\# resumes/pauses for a window in historical data}}{\text{\# of times the window occurs in the historical data}}$
   - If result is greater than the set threshold, Moneyball recommends to proactively resume/pause the database
   - Calculates and stored the probability for all windows

2. Predictive: Use ML models to predict patterns
   - If the model identifies a pause/resume pattern for a given window, Moneyball recommends to proactively start

- Predictive has better accuracy in most use cases

Moneyball - Considerations for Proactive Resume & Pause

- Savings are the computational resources that were unused as a result of pausing
- Costs are the resources used in the wait period after proactively starting
- The size of the window is an important consideration
  - Larger window results in more proactive resumes but also more wait time
  - If window is too small, the system may fail to proactively start correctly
- The length of historical data matters also
  - Settled on 3 weeks to minimize costs and reflect current usage

Moneyball - Avoiding Ineffective Pauses

- Recall that short pauses don't allow resources to be redistributed effectively & introduce additional workflows
- Two strategies to reduce ineffective pauses

1. Budgeting the number of allowed pauses per day
   - Greedy: doesn't consider length of the pause
     - Worst Case: budget used only on short pauses which don't save many resources
   - Predictive: prioritizes long pauses (based on model predictions)
     - Worst Case: budget goes unspent
   - Budgets can be specified at different granularities and intervals of time

Moneyball - Avoiding Ineffective Pauses cont.

2. Delay resource pause for a short period of time
    - Greedy: Always delay the resource pause upon logout
        - Drawback: Resources remain idle & long pauses get shortened
    - Predictive: Uses model to predict length of pause and avoid those that are shorter than the delay
        - Drawback: Model tends to predict pauses will be shorter than their actual duration meaning resources are not paused when they should be
- In practice most databases benefit from the Greedy approach with little additional cost

Moneyball - System Performance and Takeaways

- Finds middle ground between COGS & QoS
- Contributions: Predictive resume/pause & resource pause delay
- Combination of predictive resume & pause delay allow 80% of all resumes to be proactive and correct for long lived DBs
- Roughly half of all ineffective pauses are avoided
- Optimizations are used in all Azure Regions

Moneyball - Relation to Big Data Analytics

- Another step forward for serverless compute
  - Adapted to more clients/use cases
- Resource Optimization
  - Making the most of existing resources
- Reduces latency for clients and increases throughput for Azure