Colin McClelland
CPSC 324 - Final Project

Agiwal, Ankur, et al. "Napa: Powering scalable data warehousing with robust query performance at Google." *Proceedings of the VLDB Endowment* 14.12 (2021): 2986-2997.

Google Napa

Abstract

- Napa: Analytical data management system
  - ↳ principle technology: "aggressive use of materialized views which are maintained consistently as new data is ingested"
- Design principles: ① Robust Query performance
  ② Flexible config. of client DBs
- Main Challenge: Napa had to meet existing requirements
  - not a greenfield system

Introduction

- High level constraints
  - sub second query performance
  - petabytes of data
  - continuously updated
  - consistent & fresh query results
  - fault tolerant
- Began from set of requirements
  - less flexibility / not starting from clean slate
- Built to replace Mesa
  - historical data onboarded + new clients added
  - "Broader mandate" than Napa
- "Bedrock" principles of design

① Robust Query Performance
- low latency + low variance in latency regardless of load
- Napa gauntees robust query perf. w/ consistent results

② Flexibility
- not all clients have same perf. reqs. or are willing to pay

③ High throughput Data Ingestion
- implements distr table & new maintenance framework based on Log Structured Merge Tree

↳ client flexibility (what)

- Materialized views are updated consistently
  ↳ contrast to current trends of other systems which achieve perf. via efficient scans of base data
- Tradeoffs
  - coverage affects query performance
  - refresh rate affects freshness } additional associated costs
  - clients can specify both based on their needs

Design constraints

- 3 objectives: query performance, freshness, computational cost
  - Q perf: measured by latency
  - freshness: measured by time between row being added to table & being available for querying (ranges from few minutes - couple hours)
  - cost: ingestion & maintenance dominate
- Q perf must also consider predictability
- Clients must make three way tradeoff, all have diff needs
- Coupling ingestion w/ storage: new data can't be ingested until fully processed
- Coupling new data w/ querying ⇒ slows down perf
- Neither approaches work b/c
  1) freshness is sacrificed: ingestion dependent on storage bandwidth
     ↳ stale data is served
  2) Q performance consistency is sacrificed: views could be gen on the fly (lazy), but Q perf is inconsistent (fresh, but slower)

Design Choices

- Ingestion → Storage → Query Serving
- Ingestion f.w. is responsible for commiting updates (called deltas) into tables. write optimized
- Storage f.w. incrementally applies updates to tables & views
  - tables & views maintained as LSM forest
  - ea table is a collection of updates

- Compaction: deltas consolidated to form larger deltas
- table deltas get transformed into view deltas

3) Query Serving: answers client queries
   - system performs merging of table/view deltas @ query time
   - Q latency dependent on Q time merging
     ↳ faster storage can proc updates, fewer deltas need to be merged @ Q time

- Ingestion decoupled from view maintenance & view maintenance from Q proc.
  ↳ enables client to fine tune to their needs
- Low Effort: cost optimized => reduced Q perf
  ↳ translates to less aggressive compactions & more merging @ exe time
  - can also be less materialized views or reduced freshness
- High Effort: better Q perf
  ↳ low fan in merges @ Q time

▶ Providing Flexibility
- Users can specify needs which get translated into system configs
  ↳ # views, max # deltas that can be opened during proc
- System isn't static due to ingestion so Napa offers Queryable Timestamp: live marker that indicates freshness
- Now - QT = data delay
- All data up to QT timestamp can be queried by client
- Gaurantee that all data used to serve Qs will meet conditions for delivering expected Q perf
- Categories of clients:
  1) Freshness tradeoff
     - client needs good Q perf w/ moderate cost
     - Napa maintains moderate # views & fewer delta merges @ exe
     - exe fw uses fewer workers & "cheap opportunistic resources"
  2) Q perf tradeoff
     - client needs fresh data, but has lower perf standards

- system maintains less views, but more deltas can be merged
- more workers alloc for ingestion than view maintenance

3) Cost tradeoff
- client needs good perf & freshness
- low # deltas merged @ run time + more views are maintained
- many workers allocated to ensure faster ingestion, high throughput view maintenance

▶ Data Availability
- Napa gaurantees availability @ scale of data center outages
- client DBs replicated @ multiple datacenters
- the consistency across datacenters
  - decouples data & metadata ops
  - one approach: exe ingestion activity as synchronous transactions
  - Napa approach: decouple exe of data & mdata ops
    ↳ data ops exe synchronously, mdata ops used periodically to ensure replicas remain synchronized
  - QT indicates time @ which all tables & views are consistent across all data centers

▶ System Architecture
- high level arch consists of data & control planes
  - data: ingestion, storage, query serving
  - control: controller that coor. work among various sub systems
    - also responsible for sync & coor. mdata ops across datacenters
- built by leveraging existing Google infrastructure components
  - Colossus: table in Napa is collection of Colossus files
  - Spanner: functions that req specific transaction semantics (mdata)
  - F1 Query: Query serving + large scale data proc like view creation & maintenance
- Clients use ETL pipelines to insert data into tables
  - delivered to any replica & Napa incorporates to all datacenters

- Napa excels @ workloads w/ aggregations & complex filters (dashboard)
  - ↳ storage & view maintenance is key to maintain these aggregations
- low latency achieved by directing Qs to materialized views & not base tables
- low variation achieved by controlling fan in merges
- Napa tables & views are sorted, indexed, & range partitioned
  - ↳ motivated by strict latency & resource reqs which necessitate indexed key lookups
- addl partioning schemes being investigated
- Napa relies heavily on merging & sorting performance
  - ↳ sorting, merging, & GB were optimized
- Controller schedules compaction & view updates to keep count of deltas such that it can be configured
- most clients req freshness of 10s of minutes
- System has been able to maintain all client reqs wrt freshness

▶ Ingestion
- goal of i.f.w. is to allow pipelines to insert w/o much overhead
- seeks to accept w/ minimal proc & make data durable w/o being concerned w/ view maintenance
- Ingested rows get mdata timestamp for ordering & get marked as committed after meeting durability reqs
- clients can specify # tasks dedicated to ingestion

▶ Queryable Timestamp: indicates freshness of data
- only advances when ingested data meets performance requirements
- clients don't see any data after QT
- clients can easily use QT to tune performance configs
- Good Q perf: optimize for reads + ensure views available
- Table is collection of all of its delta files
- non-queryable deltas: new deltas (typically few seconds)

- deltas correspond to updates over window of time
  ↳ largest span weeks or months
- Ea delta sorted by key, range partitioned, has "B-tree like index"
- deltas merged as needed @ query time
- Underlying data is a column store, but has to manage m. views & fast lookup, so it borrows ideas from row stores like BTrees & PAX layouts in physical design
- Q latency limits max # deltas that can be merged (@ Q time)
- QT is the delta that bounds this #
- usually 10s of deltas
- limit is auto adjusted
- high perf req ⇒ lower # & vice versa
- As # inc there is a tail effect
- by keeping near constant, Napa gaurntees variance in Q latency is low
- QT dep on background ops
- there is a DB level QT (min of all table QTs)
- ea replica has a local QT which constitute the global QT
- Napa will use replicas that have a min QT based on replica availability requirements (will direct Qs to certain replicas)

Maintaining views at scale

- storage sub system is resp for maintaining views & compacting deltas
- also resp for ensuring data integrity, durability, & handling outages
- Skew in view maintenance
  - occurs in process of transforming updates on base tables into updates on the views
  - mapping of base key key space into view key space may lead to discontinuities may map to a narrow view key range
- system must auto adjust to skew to ensure that QT is not susceptible to straggler views or tables

- storage sub system adjusts to cost budgets
- Key aspects of view maintenance
  1) F1 Query as relational data pump to compact tables & maintain views
  2) Replanning to avoid skew
     - system can replan on the fly if it detects data skews
  3) Intelligence in the loop (for tail mitigation)
     - data centers are specifically chosen for task exe based on historical load, active straggler task termination, concurrent task exe

▶ Query Optimizations Challenges in view maintenance
- View maintenance system exploits data properties in the input
  ↳ sorted, partitioned (hard to recreate)
- Ex data property: <u>sort order</u> of view relative to base table
  - one approach: resort view keys based on view sort order regardless of base table sort order => @ scale: expensive
  - instead beneficial to preserve input sortedness as much as possible, even if views sort order & base table sort order only partially overlaps
- 3 classes of views (based on cost of maintenance)
  1) cheapest: those that share prefix w/ base table (ABC, AB)
  2) partial prefix w/ base table (ABCD, ABD)
     - cluster on AB, sort on D
  3) most exp: no shared prefix (ABCD, DCA)
     - req repartitioning & re-sorting
- for views w/ high cardinality reduction:
  - preserving sort order isn't as important (resort if needed)
- w/ low cardinality reduction (similar size as table)
  - sort & merge becomes impt
- Napa employs "state of the art" sort library

▶ Mechanics of compaction
- compaction combines mult. input deltas into single output delta
- improves Q perf & reduces storage consumption by
  1) sorting input rows together
  2) aggregating mult updates to same rows
- Compacting is expensive for high ingestion tables
  - reduce data freshness by delaying when data becomes queryable
- compaction is merge sorting b/c delta files are sorted individually
- fan in kept large to minimize tree height
- up to 1000 inputs before merge perf dec.

▶ Robust Query serving performance
- fast Q results still req even for large workloads
- ▶ Reducing Data in the critical path
  - uses views whenever possible insted of base table
  - pushes down filters & partial aggregations to min transfer to F1
    ↳ Napa storage & F1 Q workers not always in same data center
  - maintains sparse Btree indexes on stored data & uses this to partition input Qs into thousands of sub queries
- ▶ Min. Sequential I/Os
- common to hit high latency if data has to be read from disk
- Napa uses QT to det which mdata to use which det what data is processed. Napa ensures all mdata served from mem
- All data reads go through transparent distr data caching layer
  ↳ shares work of subqueries to ensure overlapping reads are only processed once
- Online & offline prefetching are performed to further reduce I/Os
  - offline: newly ingested data is cached for frequently Q'd tables
  - Online: "Shadow Q executor" runs ahead of main executor by skipping proc. & shares data access pattern
    ↳ more accurate prefetching

▸ Combining small I/Os
- Napa parallelizes Qs I/O calls across deltas & Q'd columns
- However, high parallelism can lead to tail latency
- Napa uses QT to limit # deltas & combines small I/Os by
  1) Lazy merging across deltas
  2) Size Based Disk Layout
     - Napa supports multiple disk layout options based on delta size
     - PAX - small deltas : good for lookup queries
     - column by column - large deltas :
       - efficient for scans, but req 1 I/O per column for lookup
     - columnar benefits w/ reduction in I/O ops

▸ Tolerating Tails & Failures
- focus on tolerating tail latency rather than eliminating it
- Non-streaming RPC:
- hedging : secondary RPC sent after a given delay & waits for faster reply
- Streaming: Napa est its progress rate & req server exe-ing to report progress - if it falls behind, new streaming RPC is started on diff server w/o losing progress
- For data center wide issues, Qs are routed to other data centers

▸ Production Metric Insights
- billions Qs a day & trillions of rows ingested

▸ Views & QT help achieve robust Q perf
- reading from views reduces variance & improves Q perf + tail latency
- plateau @ 8 views as this is point where base tables no longer needed
- reducing # deltas @ Q time
  1) less small parallel I/Os which cause tail latency
  2) data is premerged (compacted)

▸ Handling Infrastructure Issues (see figure 9)
- Q perf. remains stable despite ingestion changes or outages

▶ Client Workloads (previously discussed)
- see 9.3

▶ Related Work
- Napa provides continuous ingestion w/ high perf & tunable freshness
- further adv idea of disaggregation
- Tradeoff btwn freshness or higher cost (tunable params)
- Napa is fully indexed sys optimized for key lookups, range scans, incremental maintenance of indexes on tables & views
- supports wide variety of Qs
- drop in replacement to Mesa
  ↳ sig improved Q latency & costs to run
- Napa use B tree variant LSM tree, which are optimized for updates
- LSM trades high write throughput for fast reads
  ↳ deltas solve this

▶ Conclusion
- serves dashboards, appls, internal users
- similarities to existing systems (available, scalable, etc)
- unique: materialized views, maintaining views while ingesting, tunable
- Views optimized for continuous high bw insertions using LSM forests
- QT is live indicator of performance
- serves clients w/ diff needs

Poppe, O., Guo, Q., Lang, W., Arora, P., Oslake, M., Xu, S., & Kalhan, A. (2022, January). Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. VLDB 2022, 1279–1287. Retrieved from https://www.microsoft.com/en-us/research/publication/moneyball-proactive-auto-scaling-in-microsoft-azure-sql-database-serverless

Moneyball

Abstract

- Serverless compute currently auto scales based on workload demand, but only reactively
  - ↳ resources may not be immediately available when customer comes back online after idle period
- Goal is to predict pause/resume patterns to reduce delay + avoid taking away resources for short idle periods

Intro

- customers charged per second
- serverless can have delays, so for time sensitive appls. provisioned compute may be more suitable
- goal is to proactively resume based on historical resume patterns + avoid freeing resources for short pauses
- Challenges
  1) large search space of tunable params
     - too many to search exhaustively
     - identify trends of how params affect results to chose reasonable set
     - param choice involves tradeoff btwn QoS & COGS
  2) Opposing optimization objectives
     - 2 goals (proactive resumes & avoid short pauses) are opposing b/c incorrectly predicted resumes will inc short pauses
     - reducing short pauses will make predictions harder b/c fewer hist. resumes
  3) Changed resource usage patterns
     - "negligible" % of provisioned or serverless DBs follow strict pattern
     - provisioned is short lived & underutilized, serverless is opposite
- Moneyball seeks to find middle ground btwn proactive resume & avoiding short pauses
- Make recomendations to tune key params + reduce costs of incorrect resumes until they are used

- To reduce short pauses, 2 solutions considered:
  1) put cap on # pauses per DB / day
  2) wait time before scale down (greedy & predictive considered)
- Introduces visual component to compare effect on QoS & COGS
- transfer lessons learned from provisioned
- results span half year in 10s of Azure regions => generalizable
- Up to 80% proactive resumes are correct & 99% long lived DBs benefit
- avoid short pauses w/ logical pause (wait time). Avoids up to half of pauses & 49% of DBs benefit

▶ Moneyball problem
- Provisioned results in underutilization & extensive idle periods unless manual scale down occurs
- Manual scale down is labor intensive, expensive, error prone
- If online, serverless scaling usually has low latency b/c some resources remain alloc.
- If paused latency increases -- larger performance hit
- Current Approach is to resume / pause on login / logout
- Correct Proactive Resume: Correct if resources of a DB are used w/in a window
- aim to min operational costs assoc w/ proactive resume
- short pauses are ineffective b/c resources can effectively be reused
- Ineffective: ineffective if pause falls w/in given duration
- If pauses do not occur for long idle periods, resources are wasted
- Goals: 1 Max # correct proactive resumes 2 Min # short pauses
  3 Min op costs of these 2 optimizations

▶ Transfer learning from provisioned to serverless compute
- ▶ Features
  - half year, 10s of regions
  - variables: timestamp, DB ID, DB state, duration resumed/paused, DB compute capacity (max vCores), DB creation & deletion timestamps, region

focus on

- ML models

- Provisioned used various ML models to predict load: ARIMA, Prophet, NimbusML, Neural Nets, Exp. Smoothing, Persistent Forecast heuristic
  - used given day to predict following day
  - NimbusML was most accurate, but not much more than persist. fcast.
  - most DB fall into one of 2 categories:
    1. Easily predictable (stable load or predictable pattern)
    2. Hard to predict (load is random)
- Serverless grouped into categories
  - stable: resumed or paused @ least X% of the time in historical data
  - pattern: X% of pauses & resumes occur w/in a given window over time
  - predictable: stable or pattern
  - most serverless DBs are predictable
    - ↳ 74% stable @ least 90% of time
    - 3% follow pattern w/in 15 min @ least 90% of time
    - 23% are unpredictable
  - trained ML models on 3 weeks of data & pred a given day
  - measure hinge loss
  - NimbusML was again most accurate

- Proactive Resume
  - Algorithms
    - Example DB is unpredictable, but there is a predictable recurring resume
    - $\text{prob resume rec.} = \dfrac{\text{\# weekdays resumed in window}}{\text{\# weekdays in window}}$
    - recomended to proactively resume DB if prob is > threshold (Algo 1)
    - time complexity: O(# DBs in set • # windows / day • # rows in hist data*)

      *in the given window & set of DBs
    - reccomend to proactively resume if there @ start of window on day if there exists a resume in the predicted pattern

- same as algo 1, except has to predict patterns
- time complexity = O(# DBs · (predict function + # windows · # predicted patterns))

▶ Middle Ground btwn QoS & COGS

- total cost savings = $\sum_{s \in S}$ pauses(s) · vcores(s) · cost

- wasted cost = $\sum_{s \in S}$ wait(s) · vcores(s) · cost

- resume cost index: ratio of wasted costs to total cost savings
  ↳ depends on several tunable params (size window, length hist params)
- Long lived DB: exists for ≥3 weeks
- size of window: varied from 1-9 hours, measure % correct & wrong proactive resumes
  - % correct inc from 22 to 56 for probalistic as window grows
  - % correct inc from 63 to 80 for predictive
  - Prob benefits 25 - 62 % of DBs & pred 99%.
  - % correct resumes & benefited DB is up to 3x higher for pred than prob
  - inc window results in inc cost b/c resources are idle longer
  - Prob has up to 5x fewer wrong resumes ⟹ 5x lower cost idx
- Length of historical data: varied from 3-7 weeks
  - 3 weeks: 36% correct resumes pro & correct, 43% DB have correct proactive resumes, 12% proactive & wrong
  - percentages dec as length of hist data grows
- pred has up to 18x more wrong resumes ⟹ cost idx 10x higher

▶ Avoiding Ineffective Pauses
- short idle pauses ineffective + unnecessary workloads introduced
- avoid these by budgeting # pauses & delaying pauses

▶ Budgeting Algos
- one approach: restrict # pauses per DB/day & prioritize long pauses
- Greedy Budget: don't take the length of pause into account & just allow first k pauses

- worst case: spends budget on early short pauses & avoids later long pauses
- Wasted cost = $\sum_{s \in S}$ avoided(s) · vcores(s) · cost

- **Predictive Budget**: prioritizes pred. long pauses over pred short pauses when spending the budget
  - algo takes predictions as param & gets longest pred pauses & avoids all pauses that don't start w/ a time delta of a long pause
  - inc. overhead of making the predictions
- **Optimal Budget**: avoid top K shortest pauses per DB
  - costs less than greedy by one order of magnitude
- Predictive runs the risk of making wrong preds which results in the available budget not being spent -- avoids up to 5x more pauses than greedy
- for some cases predictive is more exp & for others it is cheaper
- Budget can be defined @ different granularities & time windows
- Cost summary:
  - Greedy order of mag higher than optimal
  - pred can't gaurantee lower cost than greedy
- Logical pause based algos:
- logical pause (logout time, resources taken away)
- physical pause [time resources taken away, log in time)
- **Greedy logical pause**:
  - pauses DB for interval of time when customer logs out
  - resources are still available in case customer returns
  - reduces delays & short pauses, but resources are idle & long pauses get shortened ⇒ greedy is expensive
  - wasted cost = $\sum_{s \in S}$ (avoided(s) + l · allowed(s)) · vcores(s) · cost
- **Predictive Logical Pause**:
  - avoids predicted short pauses w/o reducing dur. of pred long pauses
  - looks @ pred pauses & avoids those that are shorter than logical pause
  - predicts pattern per DB per day

- Optimal logical pause: avoids all pauses shorter than logical pause
- # pauses & cost depend on length of pause
- greedy & optimal algos avoid 26-70% of pauses & benefit 33-58% of DBs as dur inc from 1-11 hrs
- greedy costs up to 6x more than original
- pred pauses tend to be shorter than actual & avoids up to 19% more pauses than greedy => cost is up to 4x higher
- most DBs benefit from greedy @ low cost

Putting it all together

- Proactive Autoscale on serverless compute does good job @ summing interplay of ops
- In reactive, no proactive resume/pause so no COGS wasted, but delay inc
- introduced wait time interval to avoid ineffective pauses (logical pause)
  - wastes some COGS b/c resources are idle
- up to 80% of all resumes are proactive & correct w/in several hrs of long lived DBs & when combined w/ logical pause avoids up to half of pauses (Moneyball approach)

Related Work

- Moneyball achieves contradictory goals of enabling proactive resume & avoiding short pauses while controlling operational costs

Conclusion

- 2 optimizations introduced:
  1) reduce delays in resource availability by pred resume patterns per DB over time & proactively resume resources
  2) To reduce backend workload, avoid short pauses by logically pausing a DB that becomes idle
- results are used in all Azure regions