Procella: A Fast Versatile SQL Query Engine

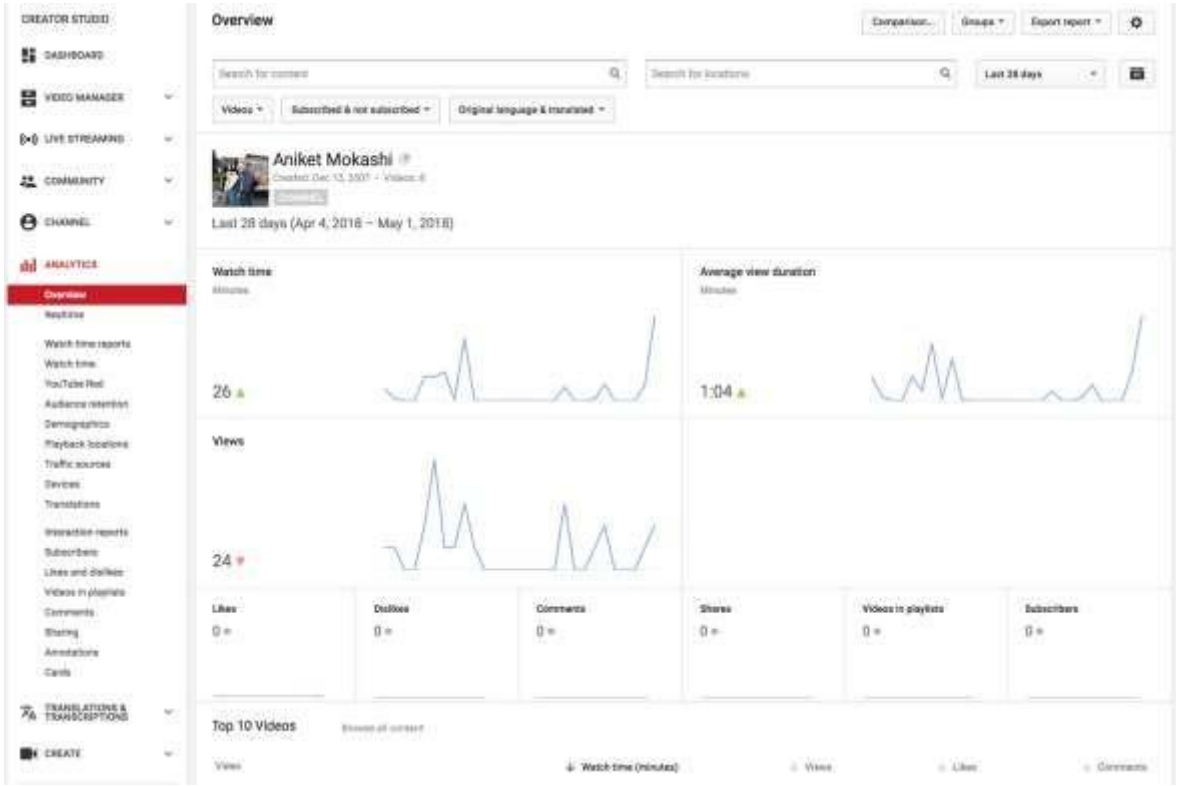
Presenter: Michael Lanthier

Slides adapted from Aniket Mokashi

Agenda

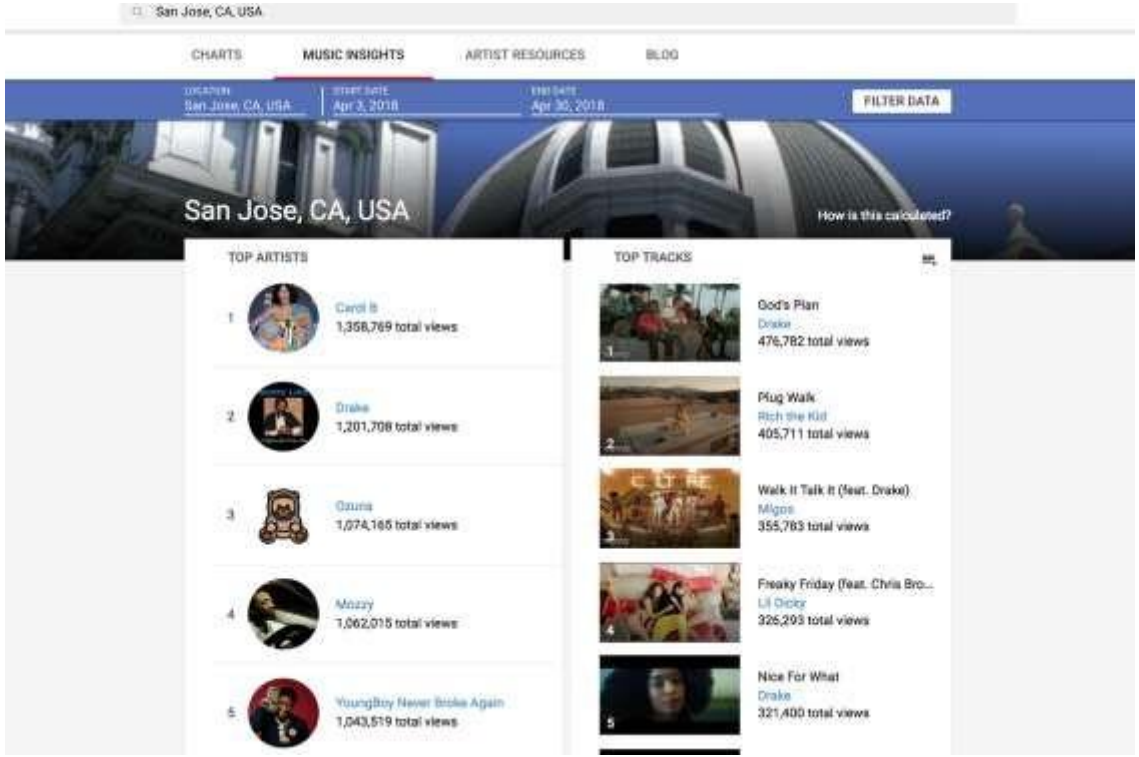
- Products at YouTube and Google using Procella
- Use cases
- Architecture
- Optimizations
- Evaluation
- Related Systems
- Comments/Questions

Youtube Analytics (youtube.com/analytics)



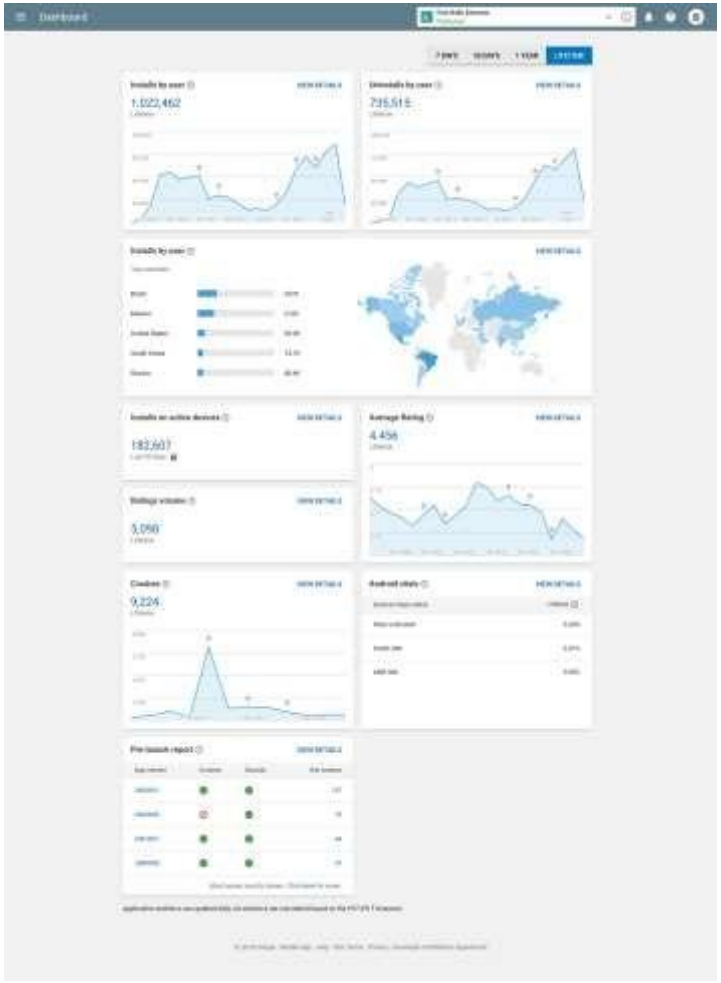
Google

Youtube Music Insights (youtube.com/artists)



And more...

- Youtube Metrics
- Firebase Performance Console
- Google Play Console
- Youtube Internal Analytics
- ...



External Reporting/Analytics

- Use cases
  - YouTube Analytics
  - Firebase Performance
  - Monitoring
  - Google Play Console
  - Data Studio
- Properties
  - High QPS, low latency ingestion and queries
  - Hundreds of metrics across dozens of dimensions
  - SQL

Internal Reporting/Dashboards

- Use cases
  - Dashboards
  - Experiments Analysis
  - Custom reporting
- Properties
  - Low QPS
  - Speed & scale
  - SQL

Real Time Insights/Monitoring

- Use cases
  - YTA Real Time (External)
  - YTA Insights (External)
  - YT Site Health (Internal)
- Properties
  - Native time-series support
  - High scan rate
  - Very high real time ingestion rate

Real Time Stats Serving

- Use cases
  - YouTube metrics (subscriptions, likes, etc.) on various YT pages
- Properties
  - Millions of rows ingested per second
  - Millions of queries per second
  - Milliseconds response time
  - Simple point queries
  - Many replicas

Ad-hoc Analytics

- Use cases
  - Data Science
  - Analytics
- Properties
  - Complex SQL queries
  - Multi-stage queries
  - Semi structured and complex data (eg - user profiles)
  - Joins

Why build something new?

A widely used SQL query engine at Google

- Moving data between multiple systems is expensive
  - Extract, Transform, and Load processes are not cheap
  - Slower time to market when adding new systems

- Reduced usability and high learning costs
  - Different APIs
  - Different languages

- Performance and scalability issues

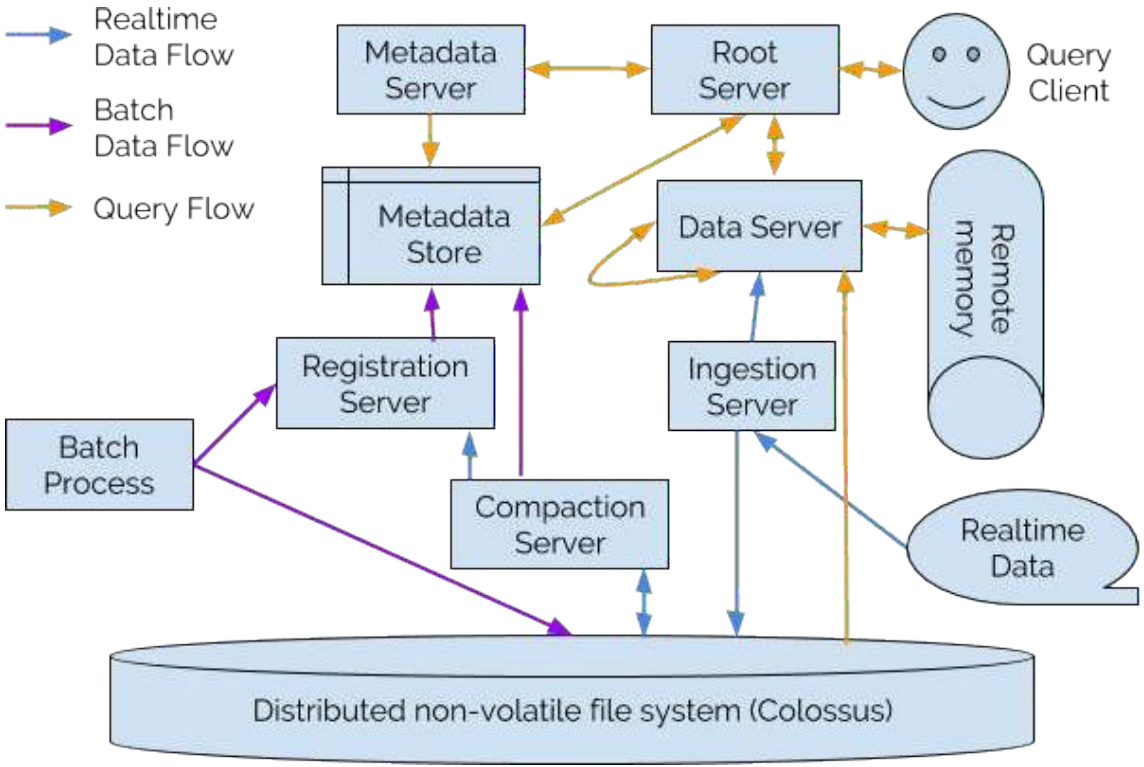
About Procella

A widely used SQL query engine at Google

- Fully featured: Most of SQL++: Structured, joins, set ops, analytic functions
- Super fast: Most queries are sub-second.
- Highly scalable: Petabytes of data, thousands of tables, trillions of rows, ...
- High QPS: Millions of point queries @ msec, thousands of reporting queries @ sub-second, hundreds of ad-hoc queries @ seconds
- Designed for real-time: Millions QPS instant ingestion, highly efficient, …
- Easy to use: SQL, DDL, Virtual tables, …

Architecture

Architecture



Google Services

- Storage using Colossus
  - Disaggregated storage
  - Immutable (Append only until finalized)
  - High cost RPCs to get data.
- Computations done on Borg
  - Supports small tasks best.
  - Machines go up and down fairly frequently so tasks must recover well.
  - Varying hardware platforms results in unpredictable performance.

Data and Metadata Storage

Data

- Organized via Tables
  - Stored in multiple files (tablets)
- Columnar format
  - Artus
  - Capacitor
- Durably stored in Colossus

Metadata

- Uses secondary structures
  - Zone maps
  - Bitmaps
  - Bloom filters ...
- Structures collected during file registration.
- Served by the Metadata Server
- Stored in the metadata store
  - Bigtable
  - Spanner

Table Management

- Standard DDL commands
  - CREATE, ALTER, DROP
- Handled by the Registration Server
  - Information then stored in the metadata store
- User can specify table information
  - Column names, data types
  - Partitioning and sorting
  - Data ingestion method

Data Ingestion

Batch

- Takes data from offline batch processes and registers it.
- Controlled by Registration Server
  - Extracts mapping of table to files
  - Creates some secondary structures
- Data servers may lazily generate expensive secondary structures.
-

Realtime

- Handled by Ingestion Server
  - Supports RPC or PubSub
- Data written to both Colossus WAL and Data Servers
  - Allows for queries to access data during writing.
  - WAL is eventually compacted to disk.
- Small files are periodically aggregated and compacted into large files

Queries

- Client connects to Root Server
  - Performs query rewrites, parsing, optimization
  - Metadata Server provides table information
- Builds a tree plan
  - Query blocks as nodes
  - Data streams as edges
    - Aggregate, Execute Remotely, Stagger Execution
  - Allows for optimization with distributed queries
- Plan is executed on other Data Servers
  - Receives plan fragments
  - Reads data from Colossus, remote memory, or local memory

Optimizations

Caching

- Aggressive caching to avoid RPCs
- Colossus metadata
  - File handles are cached to avoid open calls to Colossus name server
- Columnar file headers and footers
  - Data servers maintain a LRU cache for offset, column size…
- Columnar data and metadata
  - Expensive operations and bloom filters also are cached
  - Data holds same format in cache as in Colossus
  - Metadata servers cache information in a LRU cache
- Affinity scheduling
  - Operations on similar data goes to the same server.
  - Improves cache hit ratio

Artus File Format

- Multi-pass adaptive encoding
  - Avoids generic encoding
  - Multiple passes finds an optimal encoding
- O(log n) lookups, O(1) seeks
- Novel tree like representation of table fields
  - Prunes nested and repeated values
  - Sparse values take up little space
- Exposes dictionary, RLE information to execution engine
- Allows for rich metadata and inverted indexes
  - Encodes sorting, mins, maxs, bloom filters…
  - Allows tablets to be pruned without data being read.

Artus vs Capacitor

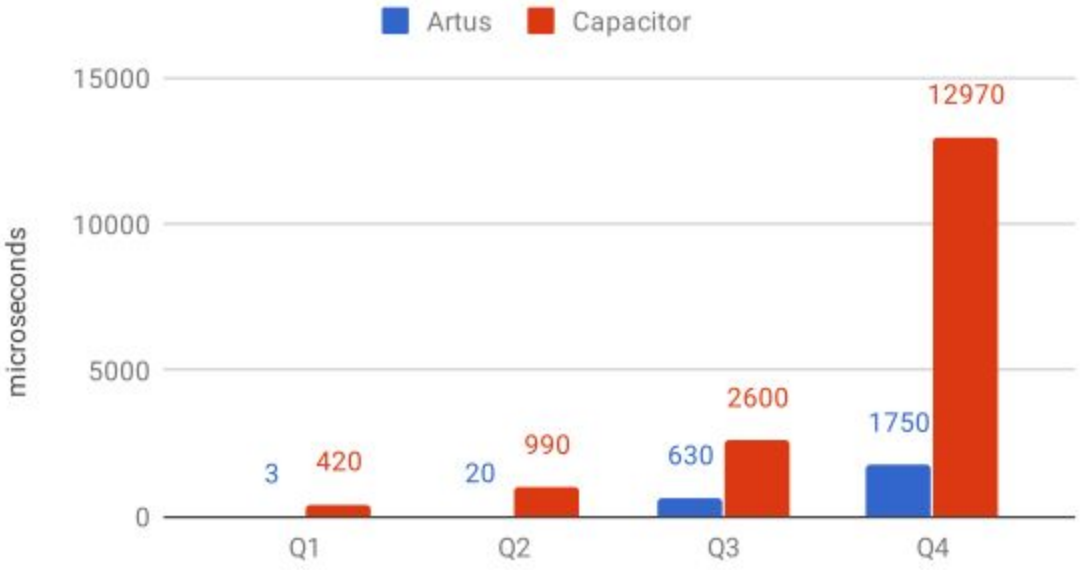


Figure 2: Artus vs Capacitor Performance.

Superluminal Evaluation Engine

- Uses extensive C++ templating
- Data processed in blocks
  - Blocks estimated to sit in the L1 cache
- Operates on native data encoding where possible.
- Does not use intermediate representations
- Filters are pushed down the execution plan

Superluminal Performance

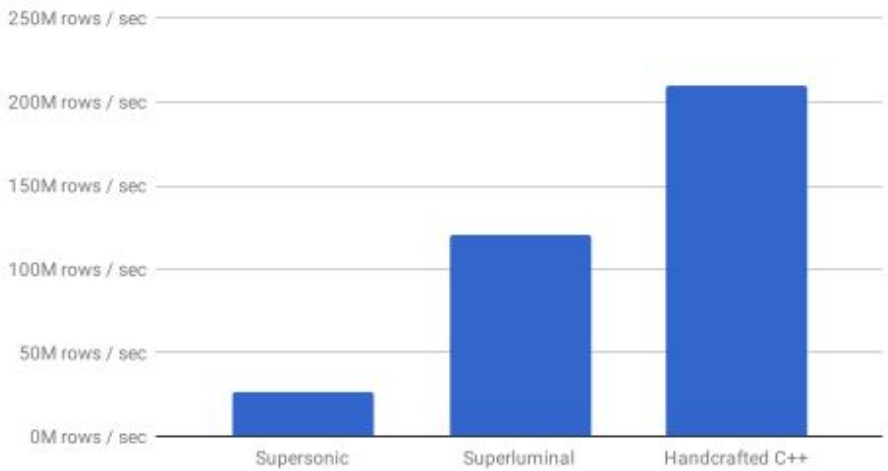


Figure 3: Superluminal vs Supersonic.

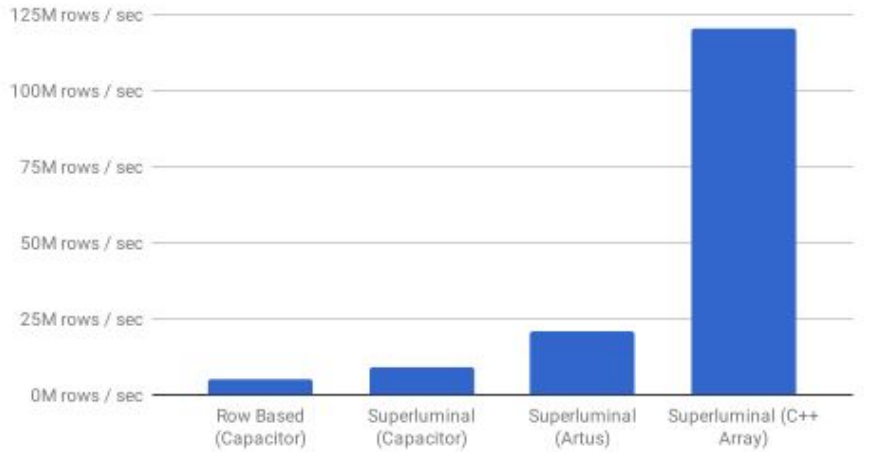


Figure 4: Superluminal performance on Artus & Capacitor.

Partitioning

- Supports partitioning and clustering.
    - YouTube partitions many tables based on date.
    - Allows for tablets to be pruned easily.
- Metadata Server retrieves and stores this information.

Distributed Joins

- Broadcast
  - One side is small enough to be loaded in memory of each Data Server running the query.
- Co-partitioned
  - Data Servers only load a small subset of data to join.
- Shuffle
  - No partitioning so intermediate servers must be used.
- Pipelined
  - One side can be reduced to a smaller set and pipelined to the other.
- Remote lookup
  - Use RPCs to get data.
  - Must push down filters to ensure that the minimum amount of data is fetched.

Tail Latency

- Root server monitors Data Server response latency during query execution.
- Slow requests are re-run on backup Data Servers.
- The number of requests to Data Servers are limited.
- Requests are given priorities.
  - Data servers handle high priority requests first.

Virtual Tables

- Generate multiple aggregates of the base table.
- Index aware aggregate selection
  - Data organization is taken into account to minimize data scans
- Supports combining batch and streaming data
- Stitched tables allow combining tables for *UNION ALL* queries

Query Optimizer

- Uses static and adaptive techniques
  - Rule based optimizer at query compile time
  - Adaptive techniques during execution time

- Statistics collected at execution time
  - Expensive to do before query occurs
  - Can be done during the shuffle step

- Adaptive techniques require overhead
  - Small queries double in time
  - Reserved for large queries

Data Ingestion

- Properly data formatting is extremely important.
  - Partitioned, clustered, sorted…

- Offline data generation tool provided
  - Artus files can be generated via MapReduce
  - Avoids expensive formatting operations

Embedded Statistics

- Serving statistics is an important use case for YouTube
  - Must hande high QPS with low latency.
  - Procella adds a statistics serving mode.

- Registered data is loaded into memory immediately.
- Metadata module is loaded onto Root Server
  - Reduces RPCs to Metadata Server
- Asynchronous updating of metadata
  - Avoid remote access to metadata at query time.
- Query plans are aggressively cached.
- Adaptive joins and shuffles are disabled.

Evaluation

Procella Scale

Property	Value
Queries executed	450+ million per day
Queries on real-time data	200+ million per day
Leaf plans executed	90+ billion per day
Rows scanned	20+ quadrillion per day
Rows returned	100+ billion per day
Peak scan rate per query	100+ billion rows per second
Schema size	200+ metrics and dimensions
Tables	100+

Procella Latencies

Property	p50	p99	p99.9
E2E Latency	25 ms	450 ms	3000 ms
Rows Scanned	17 M	270 M	1000 M
MDS Latency	6 ms	120 ms	250 ms
DS Latency	1 ms	75 ms	220 ms
Query Size	300B	4.7KB	10KB

Metadata Server Latencies

Percentile	Latency (ms)	Tablets Pruned	Tablets Scanned
p50	6	110,000	60
p90	11	900,000	250
p95	15	1,000,000	310
p99	120	33,000,000	4000
p99.9	250	38,000,000	7200
p99.99	5100	38,050,000	11,500

TPC-H queries

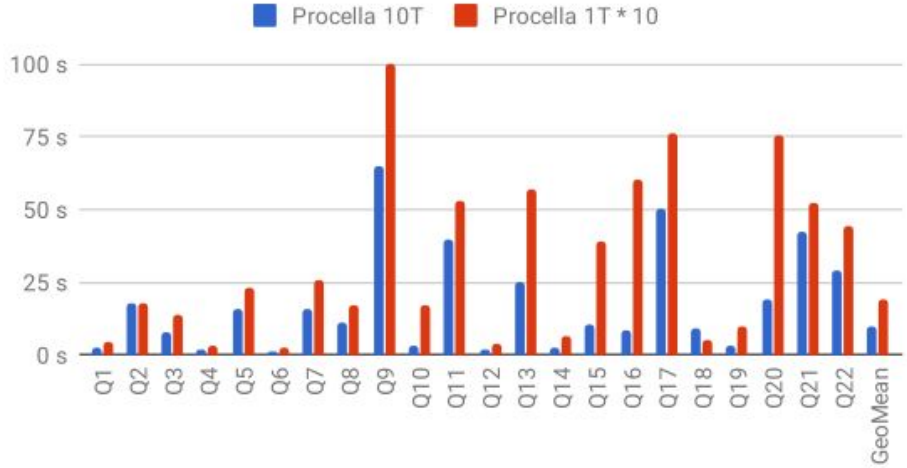


Figure 6: TPC-H Scalability - 1T Expected Execution Time vs. 10T Execution Times.

Figure 7: TPC-H 10T Per Query Throughput Execution times.

Run of TPC-H queries on:

- Static artus data
- Large shared instance
- Manually optimized queries
- 3000 core, 20 TB RAM

Related Systems

- Dremel
  - Exa-scale SQL Query Engine
  - Uses RPCs, SQL Parser, Colossus and Borg
  - Differences
    - Stateless caching
    - Non indexable file format (Capacitor)
    - Can specify when a query is “good enough”
- F1
  - Supports OLTP, ad-hoc
  - Uses Spanner as underlying storage.
  - Globally distributed and consistent

Questions/Comments

- How much extra storage is required when implementing virtual tables? Are these tables always user defined or could they be generated automatically by Procella?

- Is there a scenario where real time data will be evicted from the Data Servers buffers before it is compacted from the Colossus WAL?

- Should they have brought up network speed more?