Wildfire: Evolving Databases for New-Gen Big Data Applications


IBM
What are these New-Gen Big Data Applications?

- World has changed a lot since the 70s
  - Automating business processes → AI everywhere
- But databases are still hot
What are these New-Gen Big Data Applications?

• World has changed a lot since the 70s
  • Automating business processes ➞ AI everywhere
• But databases are still hot

And the apps want even more from the database!
  -- Higher ingest and update rates
  -- Versioning, time-travel
  -- Ingest and Update anywhere, anytime (“AP” system)
  -- More real-time analytics (HTAP)
  -- Tons of analytics

  ==> database cannot hold data in proprietary store
What are these New-Gen Big Data Applications?

- World has changed a lot since the 70s
  - Automating business processes → AI everywhere
- But databases are still hot

And the apps want even more from the database!
- Higher ingest and update rates
- versioning, time-travel
- Ingest and Update anywhere, anytime ("AP" system)
- More real-time analytics (HTAP)
- tons of analytics
  ==> database cannot hold data in proprietary store

But still want the traditional database goodies:
- Updates
- Transactions (not eventual consistency)
- Point Queries / Indexes
- complex queries (joins, optimizer, ..)
Example: Health Care

Convergence of Prevention/Monitoring (sensors on healthy people) and Cure (healthcare setting)
Example: Health Care

Convergence of Prevention/Monitoring and Cure
(sensors on healthy people)
(healthcare setting)

- High ingest rates
- Want analytics on latest readings
- Complex queries, joins, ..
- Looking for outliers => cannot drop data, need durability
- AP: cannot wait for mothership to be reachable
- Eventual consistency is a pain
  \[ V_1 \leftarrow \text{lookup}(k1); \]
  \[ V_2 \leftarrow \text{lookup}(k1); \]
  // if V1 finds match and V2 doesn’t, how to test this app?

- Lots of point queries
Wildfire Goals

**HTAP:** transactions & queries on same data
- Analytics over latest transactional data
- Analytics over 1-sec old snapshot
- Analytics over 10-min old snapshot

**Open Format**
- All data and indexes in Parquet format on shared storage
  - No LOAD
  - Directly accessible by platforms like Spark

**Leapfrog transaction speed, with ACID**
- Millions of inserts, updates / sec / node
  - Multi-statement transactions
  - With async quorum replication (sync option)
- Full primary and secondary indexing
  - Millions of gets / sec / node

**Multi-Master and AP**
- disconnected operation
- Snapshot isolation, with versioning and time travel
  - Conflict resolution based on timestamp
# Wildfire Goals

**HTAP:** transactions & queries on same data
- Analytics over latest transactional data
- Analytics over 1-sec old snapshot
- Analytics over 10-min old snapshot

**Open Format**
- All data and indexes in Parquet format on shared storage
  - No LOAD
  - Directly accessible by platforms like Spark

**Leapfrog transaction speed, with ACID**
- Millions of inserts, updates / sec / node
  - Multi-statement transactions
  - With async quorum replication (sync option)
- Full primary and secondary indexing
  - Millions of gets / sec / node

**Multi-Master and AP**
- disconnected operation
- Snapshot isolation, with versioning and time travel
  - Conflict resolution based on timestamp

**Challenge:** getting all of these simultaneously
Wildfire architecture

Applications

analytics
- can tolerate slightly stale data
- requires most recent data

high-volume transactions

spark executor

Applications

wildfire engine

shared file system

spark executor

wildfire engine

SSD/NVM
Data lifecycle

Grooming: take consistent snapshots 
resolve conflicts
Postgrooming: make data efficient for queries

- ORGANIZED zone (PBs of data)
- GROOMED zone (~10 mins)
- LIVE zone (~1sec)

OLTP nodes

TIME

Replication

Inserts, Updates, Dels
Data lifecycle

**OLTP nodes**
- HTAP (see latest: snapshot isolation)
- 1-sec old snapshot
- Optimized snapshot (10 mins stale)

**Analytics nodes**
- Bulk Load
- Lookups
- BI
- ML, etc (Spark)

**Time Zones**
- **ORGANIZED zone** (PBs of data)
- **GROOMED zone** (~10 mins)
- **LIVE zone** (~1 sec)

**LIVE zone**
- (~1 sec)
- Inserts, Updates, Dels
- OLTP nodes
- Replication

**GROOMED zone**
- (~10 mins)
- postgroom
- OLTP nodes
- Replication

**ORGANIZED zone** (PBs of data)
- HTAP (see latest: snapshot isolation)
- 1-sec old snapshot
- Optimized snapshot (10 mins stale)

**Analytics nodes**
**Live Zone**

What happens at Commit

1. append xsac deltas (Ins/Del/Upd) to common log; replicated in background
2. flush to local SSD
3. status-check if changes are quorum-visible (via heartbeats)
   -- can time-out

**AP:** Commit does not wait for other nodes; conflicts are resolved *after* commit
   (have syncwrite option for higher durability)

**Read monotonicity:** Queries always read quorum-visible state
   - Hence, *later* queries see a superset of what prior queries saw
• **Grooming is when conflicts are resolved**
  -- take quorum-visible deltas, form data blocks, and publish to shared file system
  -- groomed zone is always a consistent snapshot
• All deltas (insert/delete/update) are **upserts**: `key, (values)*, beginTime`
  • `beginTime` initialized at commit as `(localTime | nodeID)`
• No assumption about clock synchronization or speed of replication
  -- yet, we get read monotonicity
  • Idea: groom sets `beginTime ← groomTime | localTime | nodeID`
• **Conflict resolution**: versioning, based on `beginTime`
Postgrooming

Queries should run fast (BI and point)
- Compute endTime and prevRID
  - And deal with immutable storage system!
- Partition (along multiple dimensions)
- Build primary and secondary indexes

Want ready access to latest version (for the simple readers)
- Separate latest and priors

<table>
<thead>
<tr>
<th>TIME</th>
<th>ORGANIZED zone (PBs of data)</th>
<th>GROOMED zone (~10 mins)</th>
<th>LIVE zone (~1sec)</th>
</tr>
</thead>
</table>

- LATEST (key, vals*, beginTime, prevRID)
- PRIORS (key, vals*, beginTime, endTime, prevRID)

Queries should run fast (BI and point)
- Compute endTime and prevRID
  - And deal with immutable storage system!
- Partition (along multiple dimensions)
- Build primary and secondary indexes

Want ready access to latest version (for the simple readers)
- Separate latest and priors
OLAP queries via SparkSQL

- Extensions to both Catalyst Optimizer and Data Source API
- A new Spark context for SQL
- Catalyst Optimizer
  - Query HCatalog for table schemas
  - Identify plan to send to Wildfire
  - Compose a compensation plan (if needed)
- Data Source API
  - SparkSQL Logical plan → Wildfire plan
  - Plan submission to Wildfire & result passing
- Compensation plan (if needed) executed in SparkSQL
- Paper has details about pushdown analysis
POST-TRUTH
• Big data needs updates, indexes, complex queries, transactions
• AP is the reality
• PB databases will not live in proprietary storage
• It is possible to do ACID with AP
• DBMS can adopt open data formats and immutable stores – while still being fast

POST-ER-TRUTH
• Multi-shard transactions
• Serializability with AP