Piecewise Component Design: The Foundation for Supporting Pillars

Badrish Chandramouli
@badrishc

Microsoft Research
Redmond, USA
All Pillars Matter – But ...

- Our goal should be to create systems that support all pillars
  - But, allow making tradeoffs in an explicit and disciplined manner
  - Grounded on foundational principles
  - Guided by user input
All Pillars Matter – But ...

- Our goal should be to create systems that support all pillars
  - But, allow making tradeoffs in an explicit and disciplined manner
  - Grounded on foundational principles
  - Guided by user input

- Build components first, systems later
  - Do not “boil the ocean” with one monolithic entity
  - Each piece solves some specific subset of problems
  - Each piece exposes tradeoffs to users
  - Each piece has explicit non-goals
  - Pieces inter-operate seamlessly
Piece #1: Trill – a “one size fits many” engine

- Explicitly addresses
  - Query model & language
  - Single machine performance & scale
  - State checkpoint/recovery hooks

- Explicit non-goals
  - Multi-node scale out
  - SLA
  - Message reliability across nodes
  - Large state management

http://github.com/Microsoft/TrillSamples
Piece #1: Trill – a “one size fits many” engine

- Explicitly addresses
  - Query model & language
  - Single machine performance & scale
  - State checkpoint/recovery hooks

- Exposes tradeoff
  - Latency vs. throughput
    → micro-batching + columnar

- Result
  - Single QP with OOM better perf for real-time and batch datasets
  - Can be paired with various distribution platforms

- Explicit non-goals
  - Multi-node scale out
  - SLA
  - Message reliability across nodes
  - Large state management

http://github.com/Microsoft/TrillSamples
Piece #2: CRA – a generic distributed runtime

- Explicitly addresses
  - Creating logical distributed graph of processing vertices
  - Maintaining graph & connections in presence of machine failure
  - Provide hooks to vertices, to restore state and replay messages

- Non-goals
  - Query processing, app logic
  - Data plane & message delivery
  - Application state

http://github.com/Microsoft/CRA [VLDB 2016]
Piece #2: CRA – a generic distributed runtime

- Explicitly addresses
  - Creating logical distributed graph of processing vertices
  - Maintaining graph & connections *in presence of machine failure*
  - Provide hooks to vertices, to restore state and replay messages

- Exposes tradeoff
  - Choose number of vertex active copies for SLA, user controls message delivery

- Result
  - Trill + CRA = Quill (i.e., multi-node Trill) for offline datasets
  - Trill + CRA + Kafka/EventHub = Quill for real-time datasets

- Non-goals
  - Query processing, app logic
  - Data plane & message delivery
  - Application state

http://github.com/Microsoft/CRA [VLDB 2016]
Piece #3: FASTER – recoverable state management

• Explicitly addresses
  • KVS to manage large state efficiently
  • Storage (local, DFS) for state storage and recovery, caching hot data
  • Leverage temporal locality for perf (compressed WAL with in-place updates)

• Non-goals
  • Query processing
  • Message delivery
  • Multi-node distribution

http://github.com/Microsoft/FASTER
[SIGMOD 2018, VLDB 2018]
Piece #3: FASTER – recoverable state management

- Explicitly addresses
  - KVS to manage large state efficiently
  - Storage (local, DFS) for state storage and recovery, caching hot data
  - Leverage temporal locality for perf (compressed WAL with in-place updates)

- Exposes tradeoff
  - How much state are you willing to replay (or lose) after failure?

- Result
  - FASTER is orders-of-magnitude faster than today’s key-value stores & caches
  - Trill + CRA + FASTER + Kafka/EventHub → a streaming system that supports all pillars

- Non-goals
  - Query processing
  - Message delivery
  - Multi-node distribution

http://github.com/Microsoft/FASTER [SIGMOD 2018, VLDB 2018]
Responses to Panelist Questions

• All pillars matter, but
  • Design components consciously to achieve desired properties
  • Each component has explicit goals and non-goals
  • Expose tradeoffs carefully (e.g., latency vs. throughput)
  • Components inter-operate seamlessly

• Consistency matters, a lot!

• Traditional DB solutions carryover
  • But often need to be improved & extended
  • E.g., main memory – storage boundary, high level language, query semantics, ...

• We can do better than lambda!