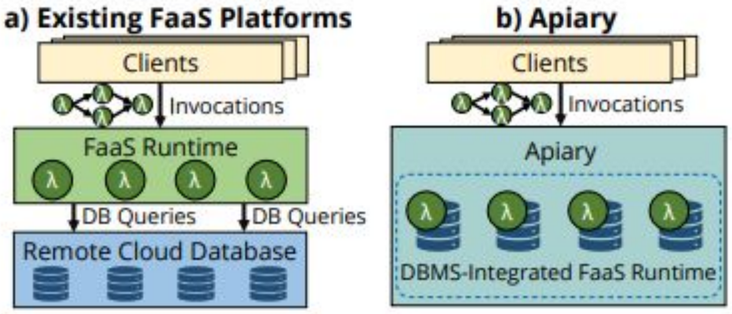
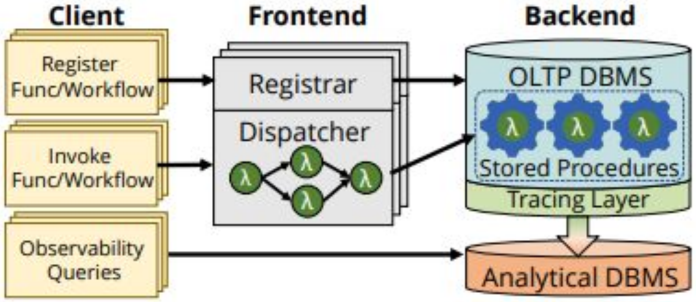
Apiary

Natan Lidukhover

Overview

- DBMS-integrated FaaS platform
- Physically and logically co-locates function execution and data management
- Strong transactional guarantees
- Exactly-once semantics
- Fault-tolerant
- Has tracing layer for observability
- Reduces communication overhead
- Designed for short-lived data-centric applications
- Relational





Interface

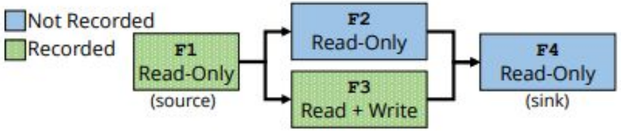
Workflow Interface

createWorkflow(List[Func], Spec)	Create a workflow from functions and a spec mapping named inputs and outputs.
groupFunctions(List[Func])	Group multiple functions into one transaction.

Function Interface

execUpdate(Query, List[Arg])	Execute a database update.
execQuery(Query, List[Arg]) → Result	Execute a database query, return its results.
returnOutput(Name, Object)	Return a named output.
retrieveInput(Name) → Object	Retrieve a named input.

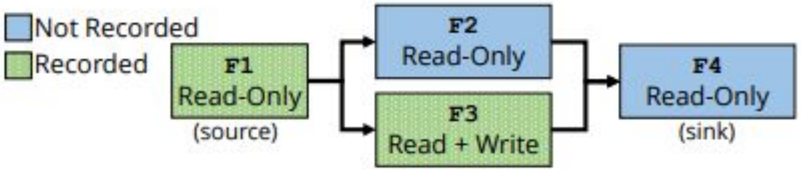
- Functions stored as stored procedures in distributed DBMS
- Functions take in and return serializable objects
- SQL queries static
- Functions deterministic
- Service calls idempotent
- Workflows directed acyclic graph



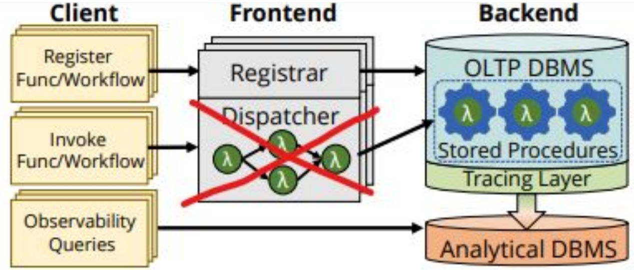

```
def checkAvail():
  query = new SQL("SELECT numAvail FROM HotelAvail
     WHERE hotelID=? AND date=?")
  inp = retrieveInput("availIn")
  avail = true;
  for (dt = inp.start; dt < inp.end; dt++):
    num = execQuery(query, inp.hotelID, dt)
    if (num < inp.numRooms):
      avail = false
      break
  returnOutput("availOut", avail)

// Omit reserve and sendEmail due to space limit.
w = createWorkflow([checkAvail, reserve, sendEmail],
  {"in": "availIn", "availOut": "reserveIn",
  "reserveOut": "emailIn", "emailOut": "out"})
w.groupFunctions([checkAvail, reserve])
```

Fault Tolerance



- Handles DBMS machine failures using DMBS
  - Replica fail-over
  - Data recovery from logs
- Handles workflow failures by recording function outputs
  - Associated with client workflow invocation using ID
  - Outputs recorded selectively using SFR algorithm (minimize overhead)
- Does not handle dispatcher failures



Algorithm 1 SFR: Selective Function Recording

function SFR(W) ▷ Input W: the workflow graph.
    $\{f_1,...,f_n\} = \text{topoSort}(W)$ ▷ f_1 is source, f_n is sink.
    $Recorded = \{\}$
    for $f_i \in \{f_n...f_1\}$ **do** ▷ Traverse from sink back to source.
        if hasWrite(f_i) **then**
            $Recorded$.add(f_i)
        else
            ▷ BFS search all recorded functions (or the sink)
            ▷ reachable without traversing a recorded function.
            $RF = BFSFindReachable(f_i, Recorded \cup \{f_n\})$
            if RF.size() > 1 **then**
                $Recorded$.add(f_i)
    return $Recorded$

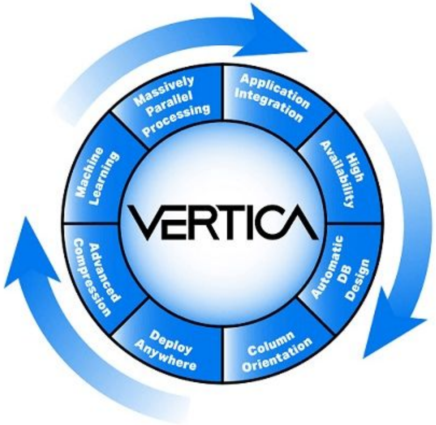
Observability

- Manual logging is expensive
- Tracing layer collects workflow information

- Collects function invocations per application

- Collects table operations within functions

- Exported to external analytical database (Vertica)



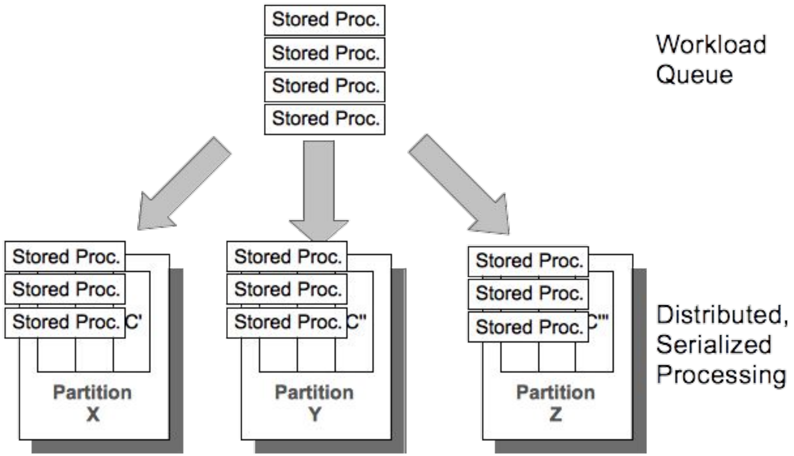

```
FunctionInvocations (func_id, timestamp,
        function_name, workflow_name, workflow_id)
```

```
TableEvents (func_id, timestamp, event_type,
        query, [record_data...])
```

DBMS

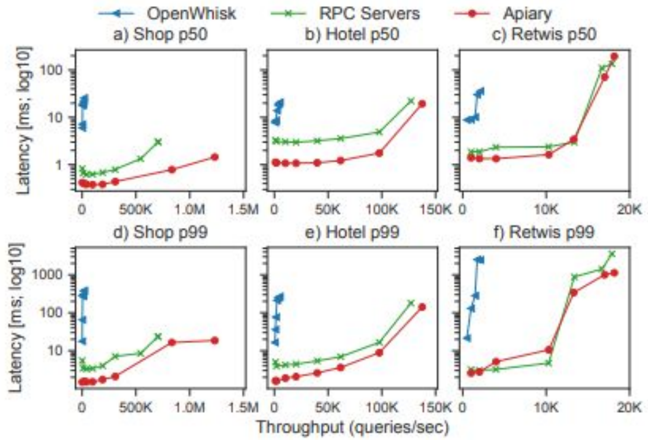
VOLTDB

- Uses VoltDB for implementation
  - ACID
  - Stored procedures support non-SQL
  - Change data capture for observability
  - Cluster resizing

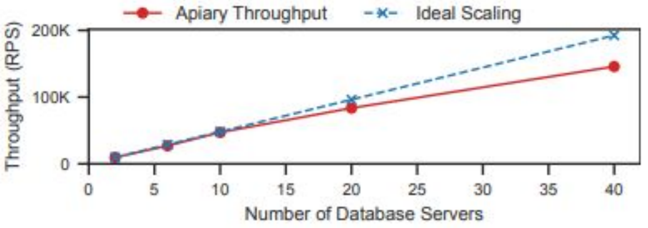


Evaluation

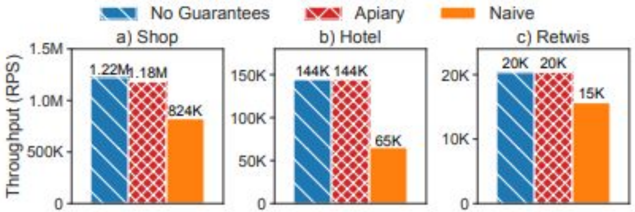
Workload	Operation	Ratio	Read-Only?	Access Rows	RPCs for μServices	# of Txns.	# of SQL Queries
Shop	Browsing	80%	Yes	8	2	1	1
	CartUpdate	10%	No	1	2	1	2
	Checkout	10%	No	5	6	3	5
Hotel	Search	60%	Yes	30	4	6	22
	Recommend	39%	Yes	1	2	1	1
	Reservation	1%	No	5	2	2	5
Retwis	GetTimeline	90%	Yes	550	3	51	51
	Post	10%	No	1	2	1	1

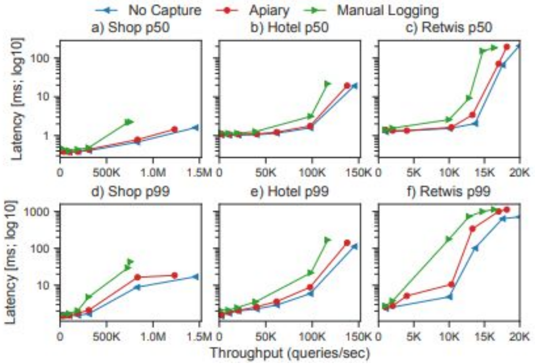


- Workloads as depicted
- OpenWhisk Java runtime has application logic for FaaS, queries external VoltDB
  - Workflows simplified to one big function
  - Apiary outperforms due to scheduling, container initialization cost, and message passing overhead
- RPC has microservice containers with application logic separate from DBMS machines
  - Apiary outperforms due to less RTT communication per DB operation
- Non-linear scaling explained away as VoltDB overhead maintaining large network

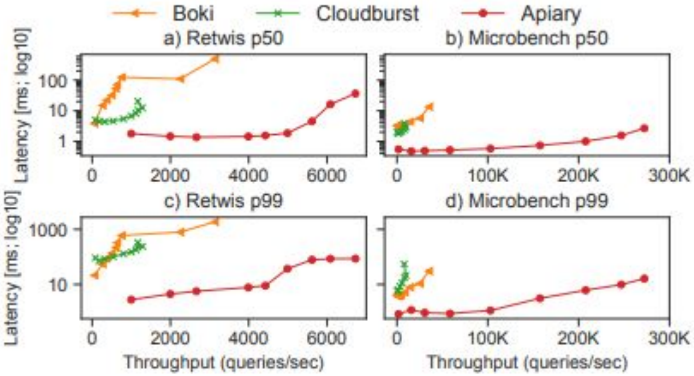


Evaluation





- SFR fault-tolerance guarantees barely affect performance
  - <5%
- Boki non-local reads when write means relatively worse when not read-heavy
- Cloudburst performance difference from more efficient local cache reads
  - Paper blames Python
  - Also no batched reads



Cost

- Low load relatively higher cost from “keeping the lights on”
- Higher load cost excels due to minimized
  - Less communication, fewer resources

System	Low Load 10 QPS	Mid Load 1K QPS	High Load 100K QPS	Mixed Load
OW + VoltDB	$1,221	$4,422	$153,956	$6,732
GCF + Firestore	**$22**	$2,679	$268,380	$4,008
Apiary + VoltDB	$917	**$917**	**$6,099**	**$3,383**

Questions?

Platform	Transactional Functions	Multi-Func. Txns.	Exactly-Once Semantics	Run-to-Completion	Data Locality
Step Functions [7]	No	No	At-Least-Once	Yes	No
Durable Functions [31]	No	No	At-Least-Once	Yes	No
Cloudburst [42]	CC	CC	No	No	Caching
FaaSTCC [27]	TCC	TCC	At-Least-Once	Yes	Caching
Hydrocache [45]	TCC	TCC	At-Least-Once	Yes	Caching
StateFun-Txns [15]	Yes	Yes	Yes	Yes	No
Beldi [47]	Yes	Yes	Yes	Yes	No
Boki [22]	Yes	Yes	Yes	Yes	Caching
Apiary	Yes	Yes	Yes	Yes	Co-location