

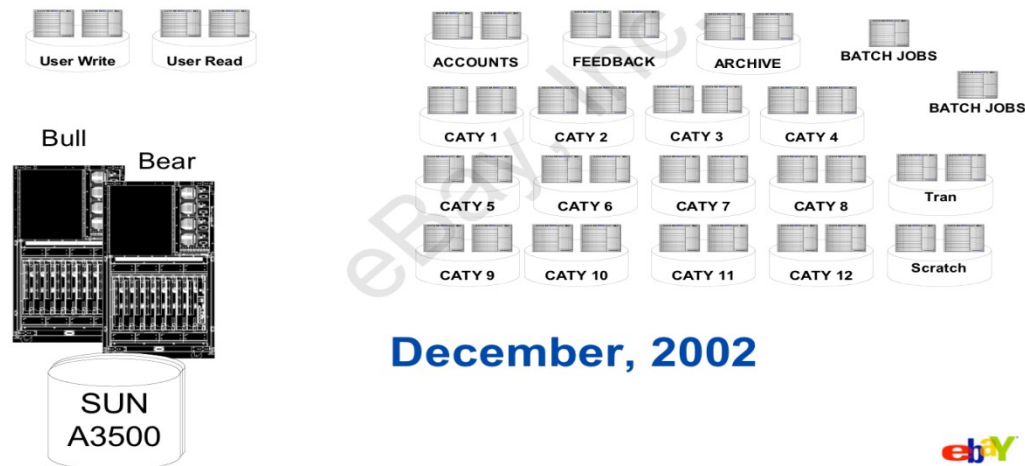
# Big Data

## Let's go back to Early-2000s...

- ✓ All the big players were heavyweight and expensive.
  - Oracle, DB2, Sybase, SQL Server, etc.
- ✓ Open-source databases were missing important features.
  - Postgres, mSQL, and MySQL.

**V2.5** April 2001 – December 2002

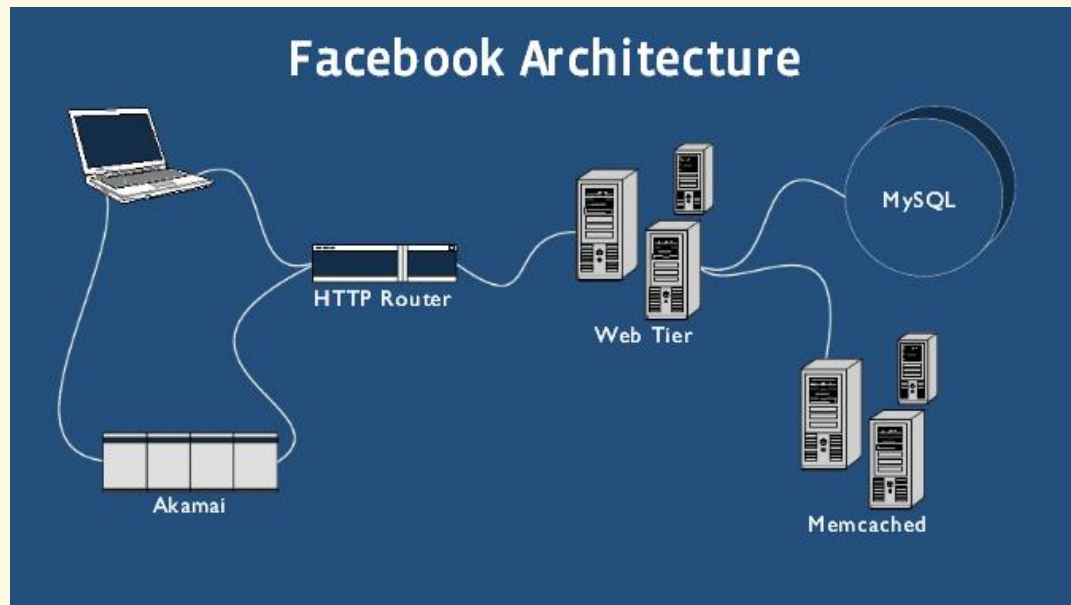
- Horizontal scalability through database splits
- Items split by category
- SPOF elimination



**December, 2002**

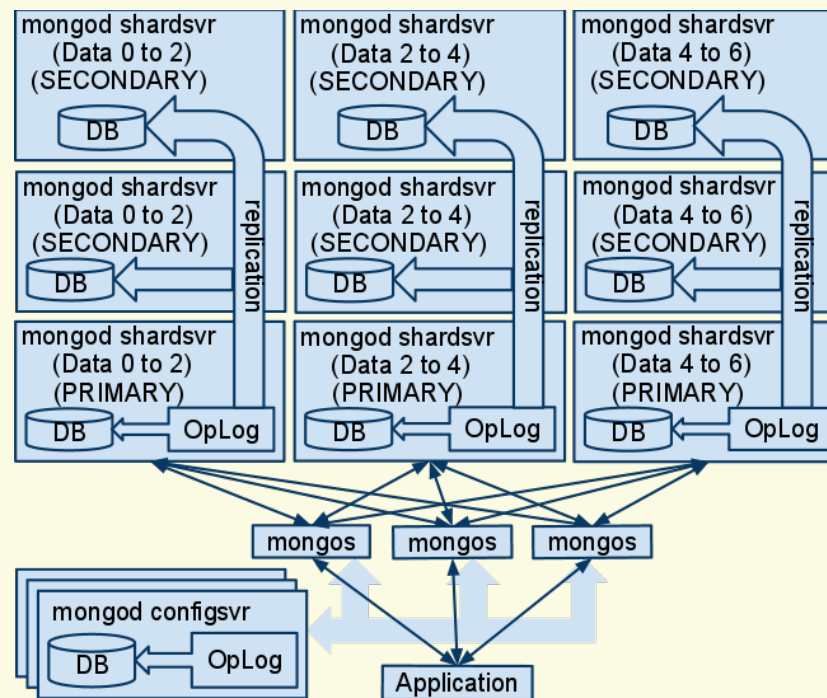
## Mid-2000s

- ✓ MySQL + InnoDB is widely adopted by new web companies:
  - Supported transactions, replication, recovery.
  - Still must use custom middleware to scale out across multiple machines.
  - Memcache for caching queries.



## Late-2000s

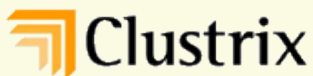
- ✓ NoSQL systems are able to scale horizontally:
  - Schemaless.
  - Using custom APIs instead of SQL.
  - Not ACID (i.e., eventual consistency)
  - Many are based on Google's BigTable or Amazon's Dynamo systems.



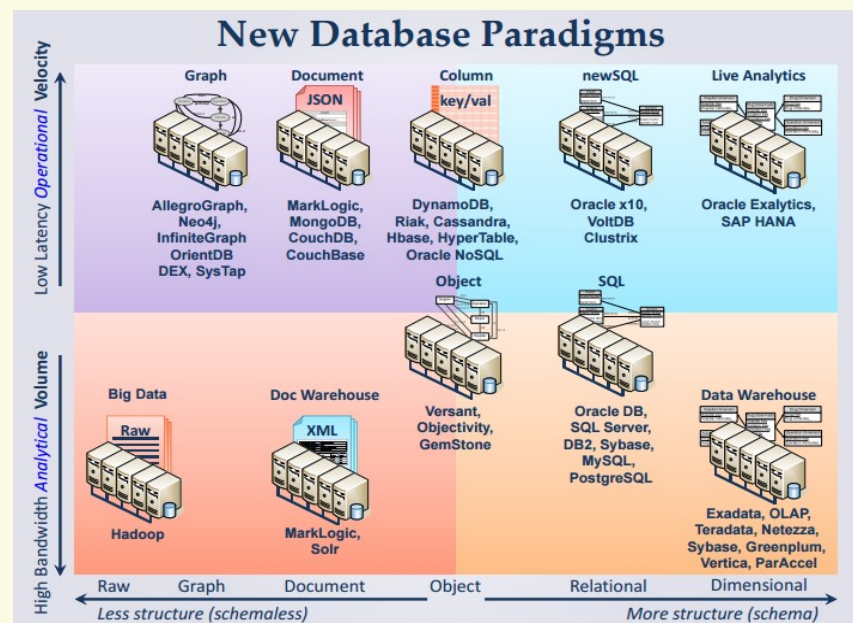
## Early-2010s

- ✓ New DBMSs that can scale across multiple machines natively and provide ACID guarantees.
  - MySQL Middleware
  - Brand New Architectures

- ✓ “New SQL”



## newSQL





## old OLTP and old SQL

- ✓ An information system can be transactional (OLTP) or/and analytical (OLAP)

- ✓ **OLTP**

transa

- ve

- da

- ef

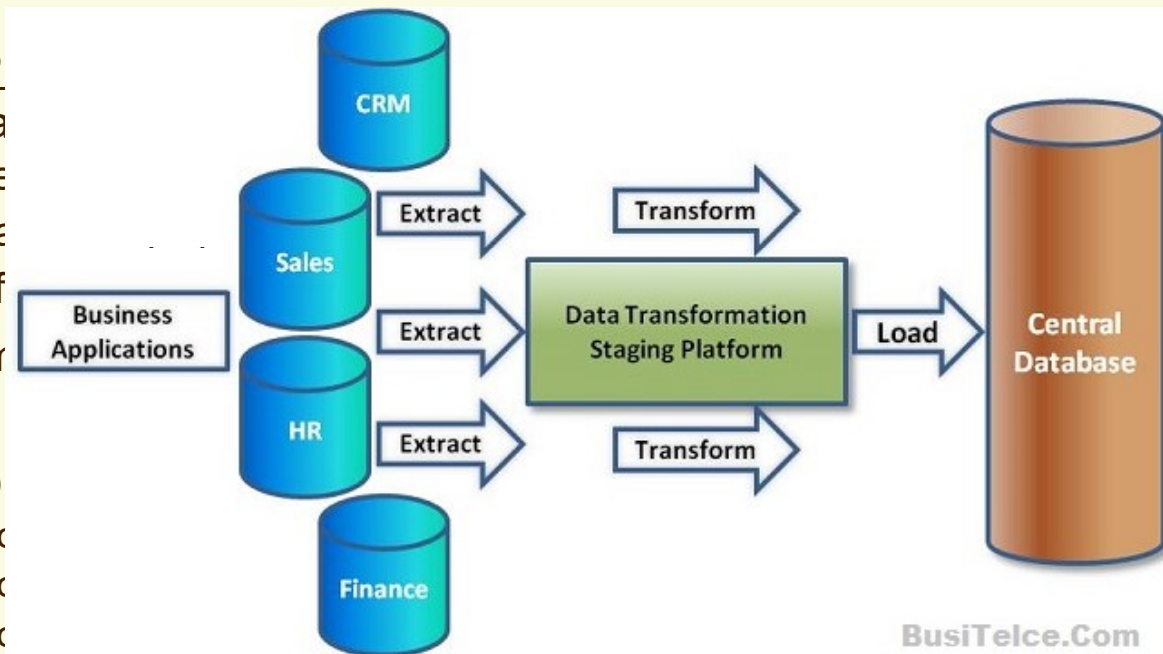
- ✓ schem

- ✓ Old O

- Co

- Co

- Co



ort on-line

- ✓ Old SQL: techs, systems and vendors supporting old OLTP

## New requirements (new OLTP)

---

- ✓ Web changes everything
- ✓ Large scale systems, with huge and growing data sets
  - ✓ 9M messages per hour in Facebook
  - ✓ 50M messages per day in Twitter
- ✓ Information is frequently generated by devices (cellphones, PDAs, sensors...)  
-> **“Online”**
- ✓ High concurrency requirements, high-throughput ACID write ->  
**“Transaction”**
- ✓ High Availability + Durability: core database requirements
- ✓ Need for high throughput
- ✓ Need for real-time analytics



## Challenge

---

- ✓ Ingest the firehose in real time
- ✓ Process, validate, enrich and respond in Real-time
- ✓ Real-time analytics
- ✓ Options:
  - Old SQL - Legacy RDBMS vendors
  - NoSQL: give up SQL and ACID
  - NewSQL: SQL + ACID + new architecture

# noSQL

- ✓ Give up SQL
- ✓ Give up ACID
  - Data accuracy
  - Funds transfer
  - Integrity constraints
  - Multi-record state
- ✓ noSQL fits
  - Non-transactional systems
  - Single record transactions that are commutative
- ✓ noSQL is not a good fit for
  - New OLTP: gaming, purchasing, order management, real-time analytical, etc

## NewSQL: informal definition

---

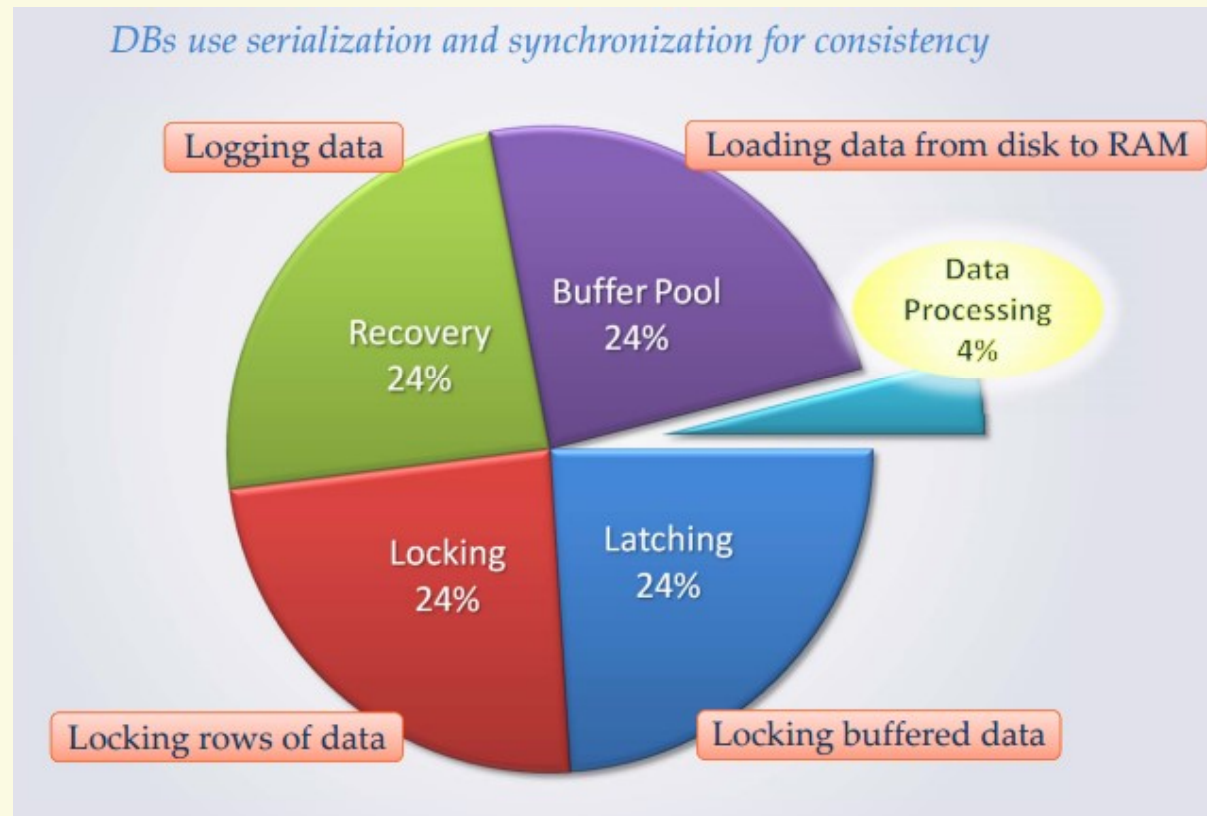
- ✓ SQL is good
  - ✓ ACID is good
  - ✓ Figure out a way to make oldSQL perform
  - ✓ Make it scale like noSQL
  - ✓ Make it available
- 
- ✓ “A DBMS that delivers the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads.”

**-- 451 Group**

## NewSQL: definition

- ✓ SQL as the primary mechanism for application interaction
- ✓ ACID support for transactions
- ✓ A non-locking concurrency control mechanism so real-time reads will not conflict with writes, and thereby cause them to stall.
- ✓ An architecture providing much higher per-node performance than available from the traditional "elephants"
- ✓ A scale-out, shared-nothing architecture, capable of running on a large number of nodes without bottlenecking
  - Michael Stonebraker

## Traditional DBMS overheads



“Removing those overheads and running the database in main memory would yield orders of magnitude improvements in database performance”

## NewSQL design principles

- ✓ SQL + ACID + performance and scalability through modern innovative software architecture
- ✓ *Principle 1: minimizing or stay away from locking*
- ✓ *Principle 2: rely on main memory*
- ✓ *Principle 3: try to avoid latching*
- ✓ *Principle 4: cheaper solutions for HA*



## NewSQL design principles

---

- ✓ new solution other than low-level record level locking mechanism
  - Transaction processed in timestamp order with no locking (voltDB)
  - multisession concurrency control (nuoDB)
  - Optimistic concurrency control (Google)
  - Principle: minimizing or stay away from locking

## NewSQL design principles

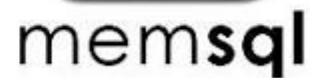
- ✓ new solution for buffer pool overhead
  - Main memory DBMS
  - Moderate case is tilted towards main memory
  - [Principle 2: rely on main memory](#)
- ✓ new solution to latching for shared data structures
  - new way to manage B-trees
  - Single-threading
  - [Principle 3: try to avoid latching](#)
- ✓ new solution for write-ahead logging
  - Built-in replication
  - [Principle 4: cheaper solutions for HA](#)

A spiral-bound notebook with a brown cover and a cream-colored page. The spiral binding is on the left side. A horizontal line is drawn across the page, and a gray rectangular box is positioned below it.

*newSQL databases*

## NewSQL: categories

- ✓ New approaches: VoltDB, Clustrix, NuoDB
- ✓ New Storage engines: TokuDB, ScaleDB
- ✓ Transparent Clustering: ScaleBase, dbShards



*voltDB*

# VoltDB

- ✓ VoltDB is an in-memory, horizontally scalable, ACID compliant, fast RDBMS
- ✓ Backed and architected by Michael Stonebraker
- ✓ An open source project
- ✓ Java + C/++
- ✓ Available in community and commercial editions



## Technical Overview

---

- ✓ VoltDB tries to avoid the overhead of traditional databases
  - ✓ K-safety for fault tolerance
  - ✓ In memory operation for maximum throughput
    - reduce buffer management
  - ✓ Partitions operate autonomously and single-threaded
    - no latching or locking
- ✓ Built to horizontally scale

## Technical Overview – Partitions (1/3)

- ✓ One **partition** per physical CPU core
  - Each physical server has multiple VoltDB partitions
- ✓ **Data** - Two types of tables
  - **Partitioned**
    - Single column serves as partitioning key
    - Rows are spread across all VoltDB partitions by partition column
    - Transactional data (high frequency of modification)
  - **Replicated**
    - All rows exist within all VoltDB partitions
    - Relatively static data (low frequency of modification)
- ✓ **Code** - Two types of **work** – both ACID
  - **Single-Partition**
    - All insert/update/delete operations within single partition
    - Majority of transactional workload
  - **Multi-Partition**
    - CRUD against partitioned tables across multiple partitions
    - Insert/update/delete on replicated tables

## Technical Overview – Partitions (2/3)

### ✓ Single-partition vs. Multi-partition

select count(\*) from orders where customer\_id = 5  
**single-partition**

select count(\*) from orders where product\_id = 3  
**multi-partition**

insert into orders (customer\_id, order\_id, product\_id) values (3,303,2)  
**single-partition**

update products set product\_name = 'spork' where product\_id = 3  
**multi-partition**

Partition 1		
1	101	2
1	101	3
4	401	2

1	knife
2	spoon
3	fork

Partition 2		
2	201	1
5	501	3
5	502	2

1	knife
2	spoon
3	fork

Partition 3		
3	201	1
6	601	1
6	601	2

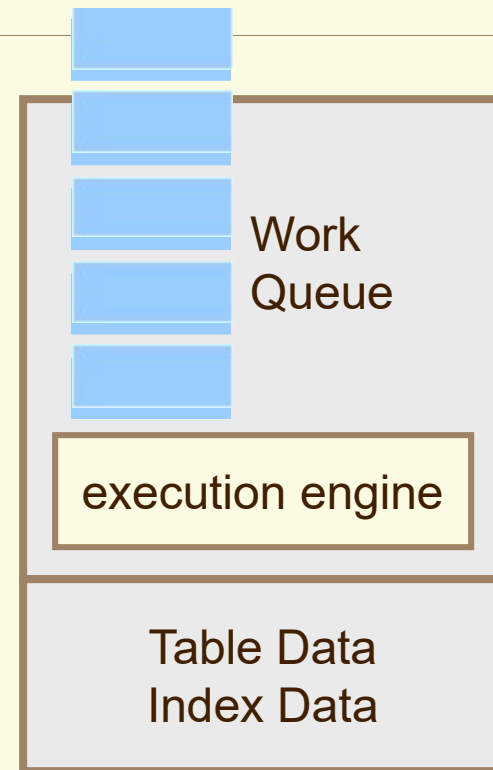
1	knife
2	spoon
3	fork

table orders :    customer\_id (partition key)  
(partitioned)    order\_id  
                     product\_id

table products :    product\_id  
(replicated)        product\_name

## Technical Overview – Partitions (3/3)

- ✓ Looking inside a VoltDB partition...
  - Each partition contains data and an execution engine.
  - The execution engine contains a queue for transaction requests.
  - Requests are executed sequentially (single threaded).



- Complete copy of all replicated tables
- Portion of rows (about 1/partitions) of all partitioned tables

# Technical Overview – Compiling

- ✓ The database is constructed from
  - The schema (DDL)
  - The work load (Java stored procedures)
  - The Project (users, groups, partitioning)
- ✓ VoltCompiler creates application catalog
  - Copy to servers along with 1 .jar and 1 .so
  - Start servers

## Schema

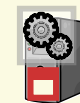
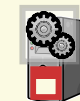
```
CREATE TABLE HELLOWORLD (  
  HELLO CHAR(15),  
  WORLD CHAR(15),  
  DIALECT CHAR(15),  
  PRIMARY KEY (DIALECT)  
);
```

## Stored Procedures

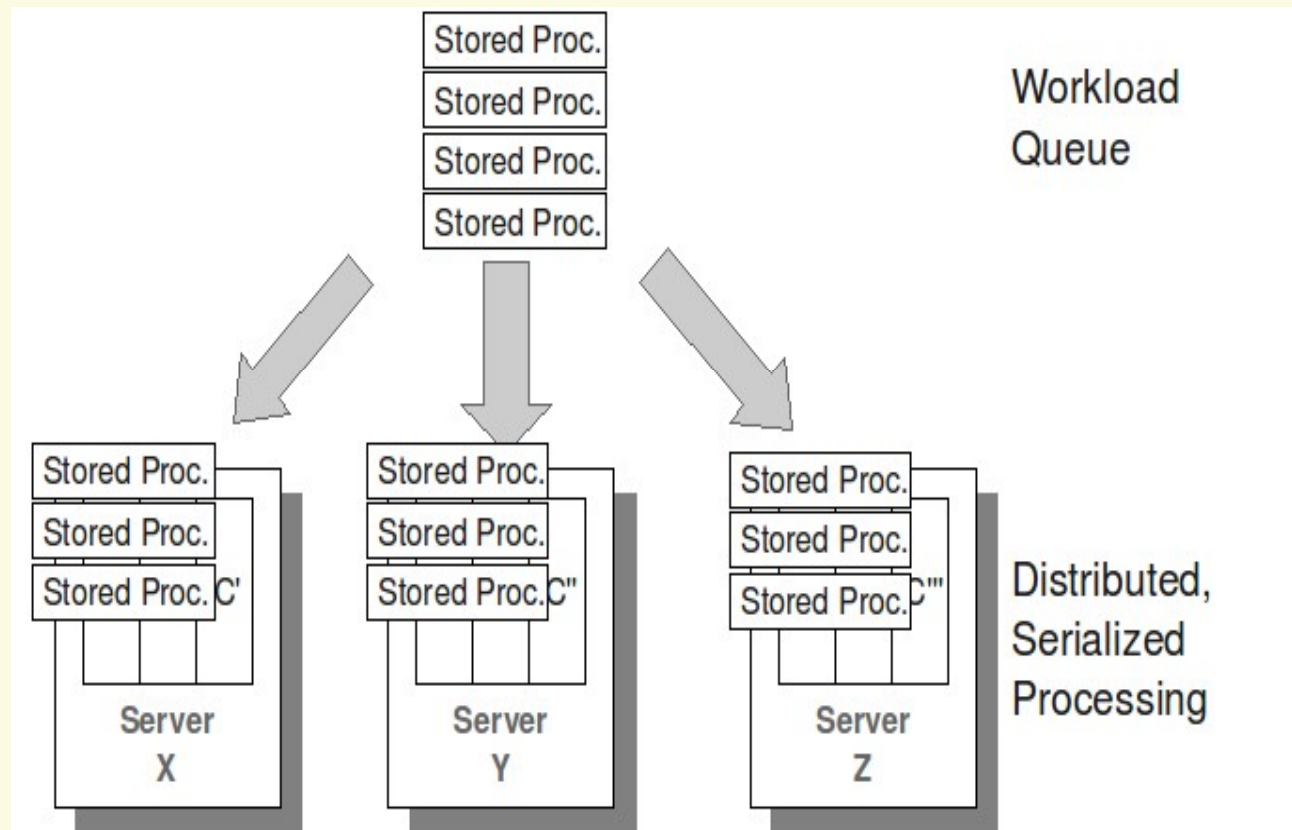
```
import org.voltodb. * ;  
  
@ProcInfo(  
  partitionInfo = "HE  
  singlePartition = t  
  
public final SQLStmt  
public VoltTable[] run
```

## Project.xml

```
<?xml version="1.0"?>  
<project>  
  <database name='data'  
    <schema path='ddl.'  
      <partition table=''  
    </database>  
</project>
```



## Transaction Model

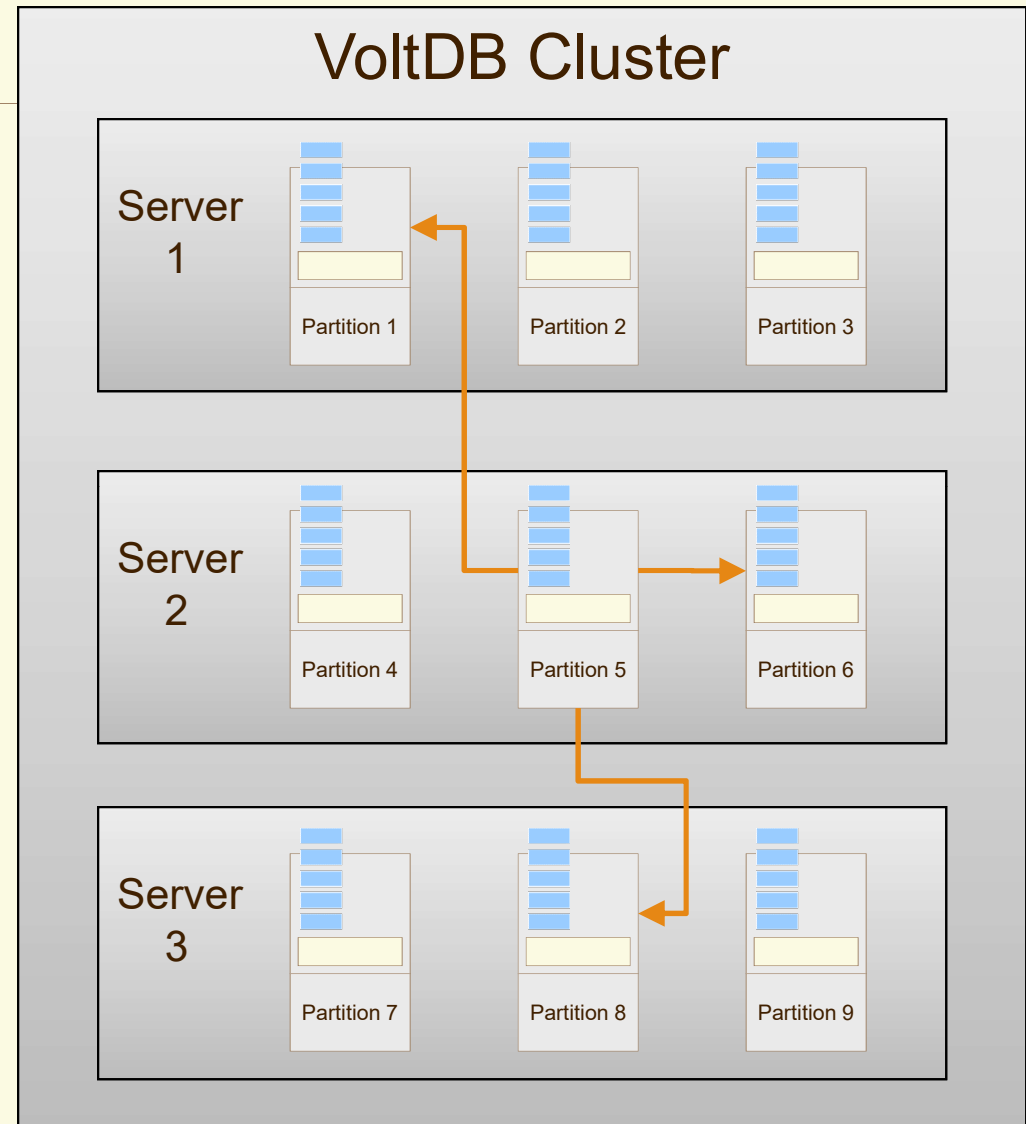


Procedures **routed** to, **ordered** and **run** at partitions



# Transaction Execution

- ✓ Single partition transactions
  - All data is in one partition
  - Each partition operates autonomously
- ✓ Multi-partition transactions
  - One partition distributes and coordinates work plans



## Data Availability and Durability

---

### ✓ High Availability

- Data stored on server replicas (user configurable)
- Failover data redundancy
- No single point of failure

### ✓ Database Snapshots

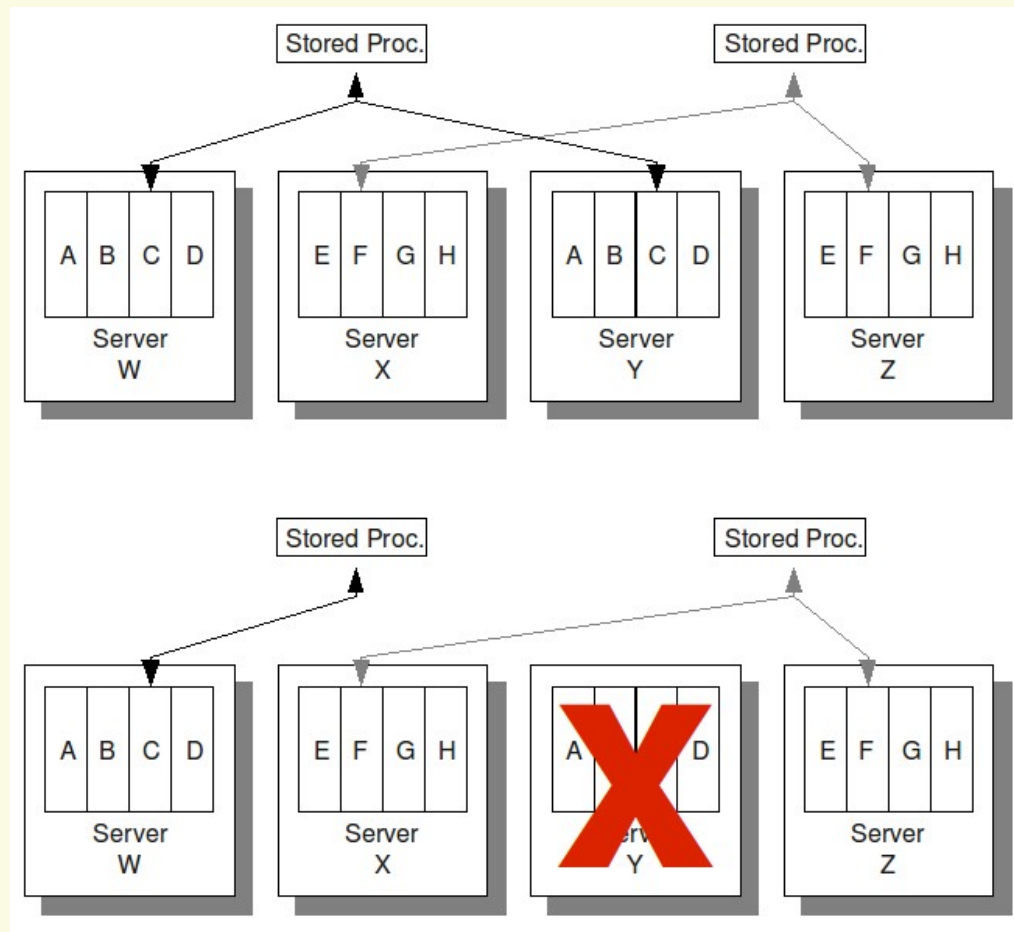
- Simplifies backup/restore
- Scheduled, continuous, on demand
- Cluster-wide consistent copy of all data

### ✓ Command Logging

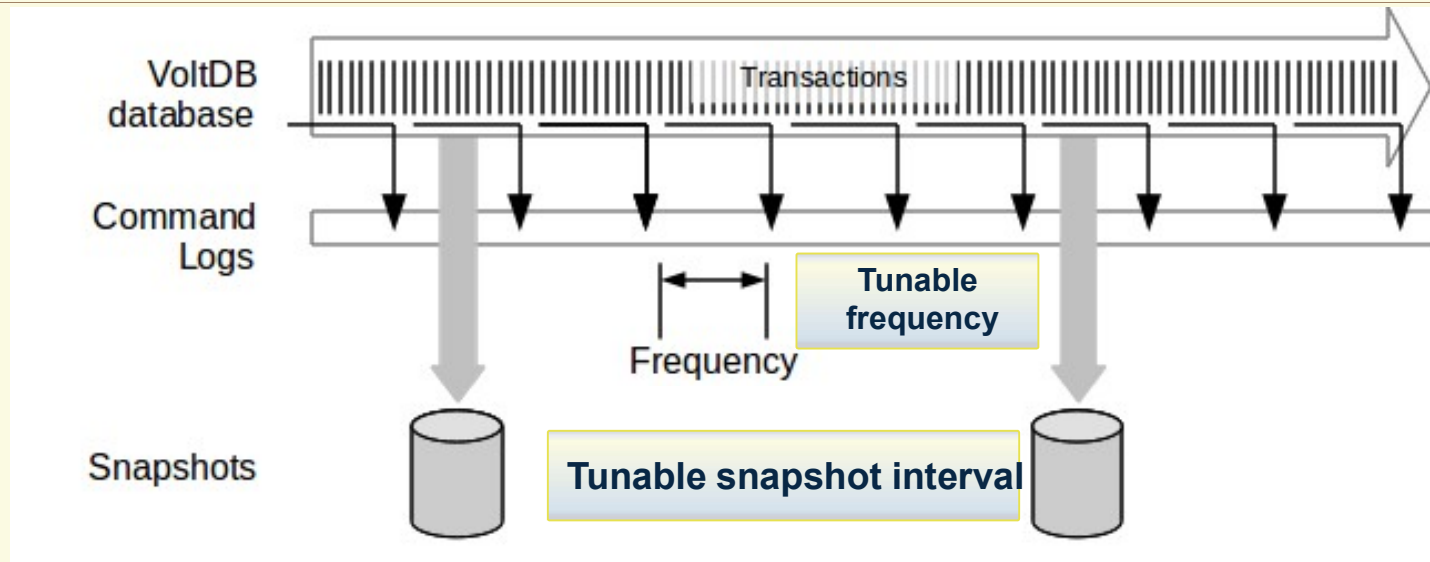
- Between Snapshots, every transaction is durable to disk

# K-safety

- ✓ Duplicate database partitions for fault tolerance. K: # of replicas



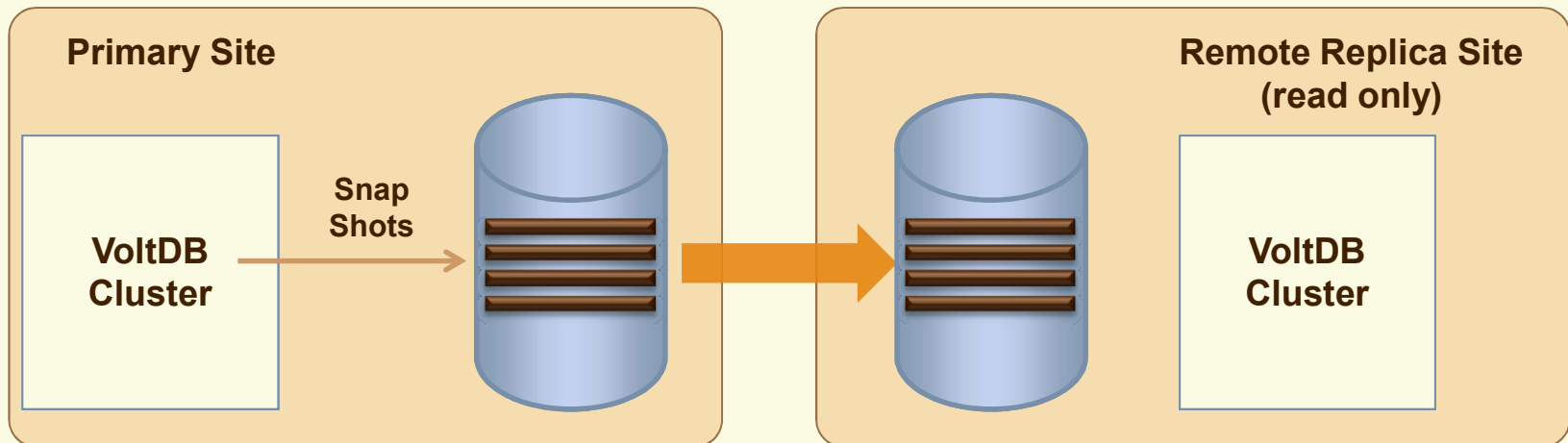
# Command Logging



- Synchronous logging provides highest durability at reduced performance
- Asynchronous logging best performance at reduced durability

# Disaster Recovery

- ✓ Disk snapshots replicated via storage system
- ✓ Stream command logs from Primary to Replica
- ✓ Run from Replica on DR event, reverse on recovery



## Lack of concurrency

- ✓ Single-threaded execution within partitions (single-partition) or across partitions (multi-partition)
- ✓ No need to worry about locking/dead-locks
  - great for “inventory” type applications
    - checking inventory levels
    - creating line items for customers
- ✓ Because of this, transactions execute in microseconds.
- ✓ However, single-threaded comes at a price
  - Other transactions wait for running transaction to complete
  - Useful for OLTP, not OLAP



*nuoDB*

## nuoDB

---

- ✓ nuoDB is an elastically scalable, ACID compliant, 100% SQL newSQL Database
- ✓ Backed and architected by Jim Starkley
- ✓ Runs on JVM
- ✓ Proprietary source project

# NuoDB: Architecture



- ✓ Multi-tier Architecture
  - Transaction tier
  - Storage tier
  - Management tier
- ✓ Multi-Tenant
- ✓ Heavy use of memory
  - hot data stays in memory
  - Cold data in persistent store
- ✓ Object Oriented
  - Objects are atoms
- ✓ Asynchronous Messaging
- ✓ Partial, On-Demand replication
- ✓ MVCC - Concurrency

# Technical Overview – Tiered Architecture

## ✓ Tiered Architecture

### – Transactions: Transaction Engine

- Parse, compile, optimize and execute SQL commands
- Stores some information in memory locally
- Map to locate the information
- Any transaction engine can service any piece of information regardless of where it resides
- Adding transaction engines -> More throughput

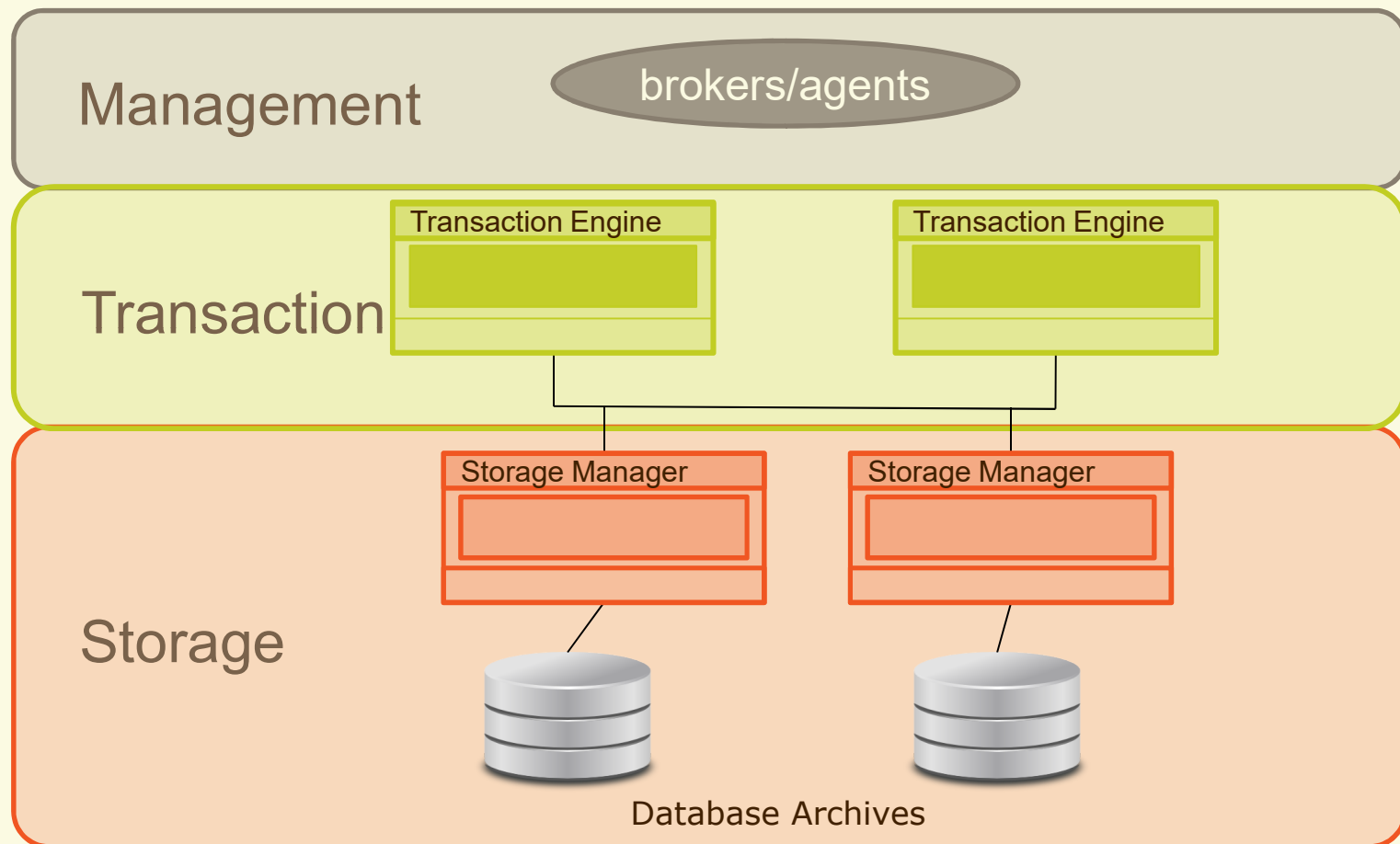
### – Storage: Storage manager

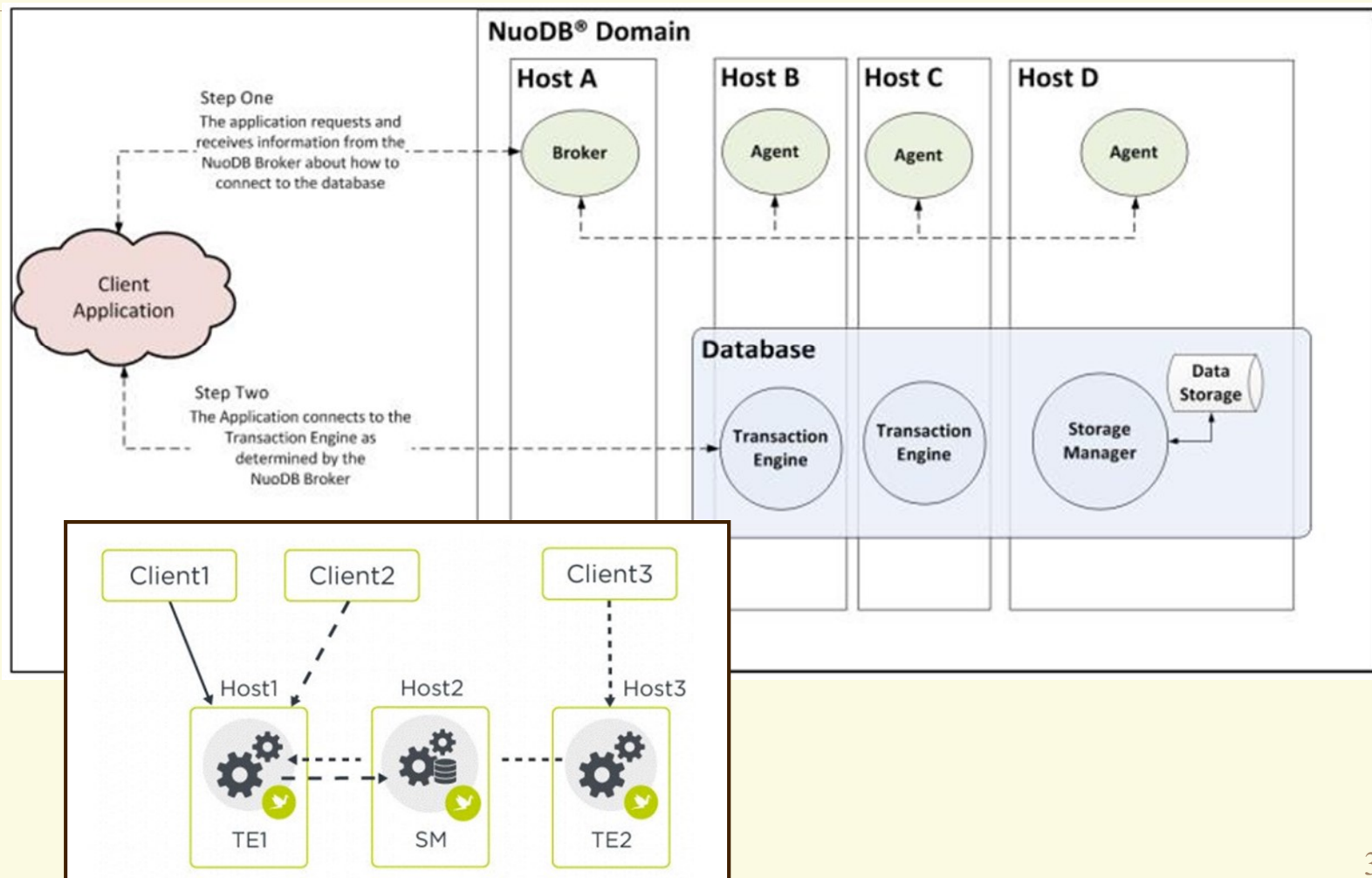
- Can run on HDFS or Amazon S3
- Adding storage manager -> more resistance to failure

### – Management: Agents and Brokers

- (Re)starting transaction engines and storage managers
- Collect statistics from them
- Clients connect to TE via Brokers

# Multi-tier architecture





## Conclusions

---

- ✓ NewSQL is an established trend with a number of options
- ✓ Hard to pick one because they're not on a common scale  
No silver bullet
- ✓ Growing data volume requires ever more efficient ways to store and process it

# oldSQL, NoSQL and NewSQL: pros/cons

## ✓ OldSQL

- +: proven tech and standard SQL, ACID
- +: rich client ecosystem
- +: established
- -: not a scalable
- -: complex

## ✓ NoSQL

- +: higher availability
- +: support
- -: fundamental
- -: require a

## ✓ NewSQL

- +: stronger
- +: familiar
- +: richer architecture
- +: leverage
- -: no NewSQL is as general-purpose as traditional SQL systems
- -: in-memory architecture may hinder the application for large dataset
- -: partial access to the rich tooling of traditional SQL

