DS256:Jan17 (3:1)

# NoSQL Databases

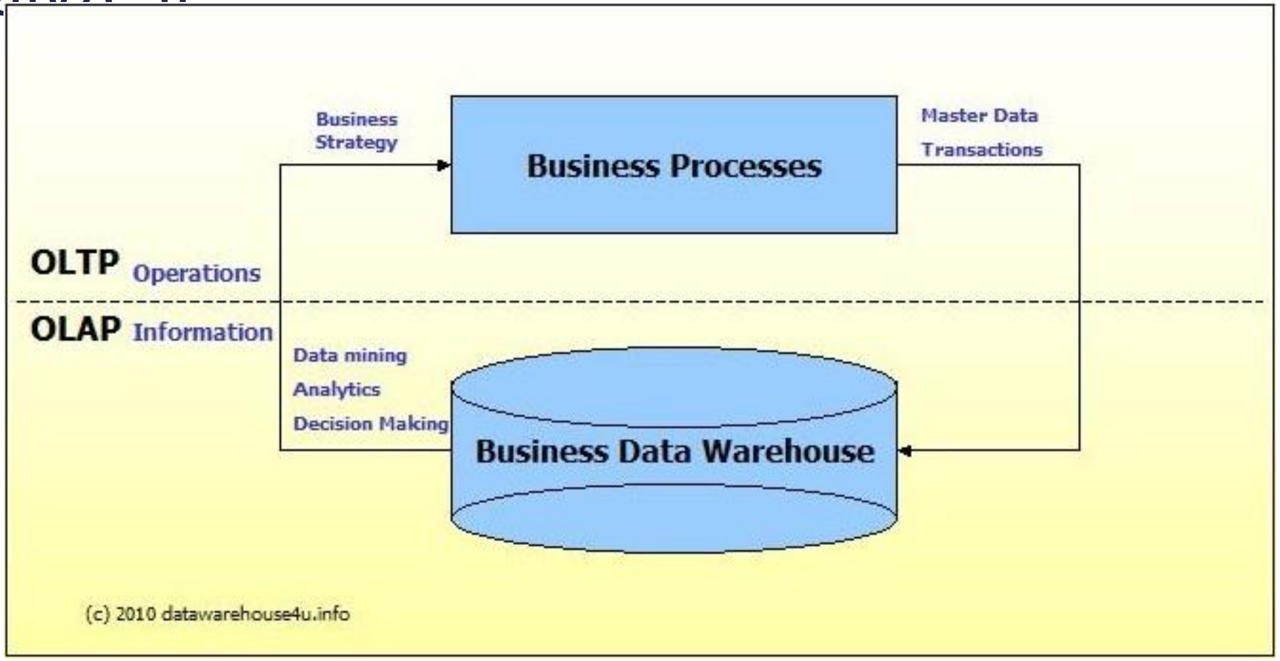
Yogesh Simmhan

28 Mar, 2017

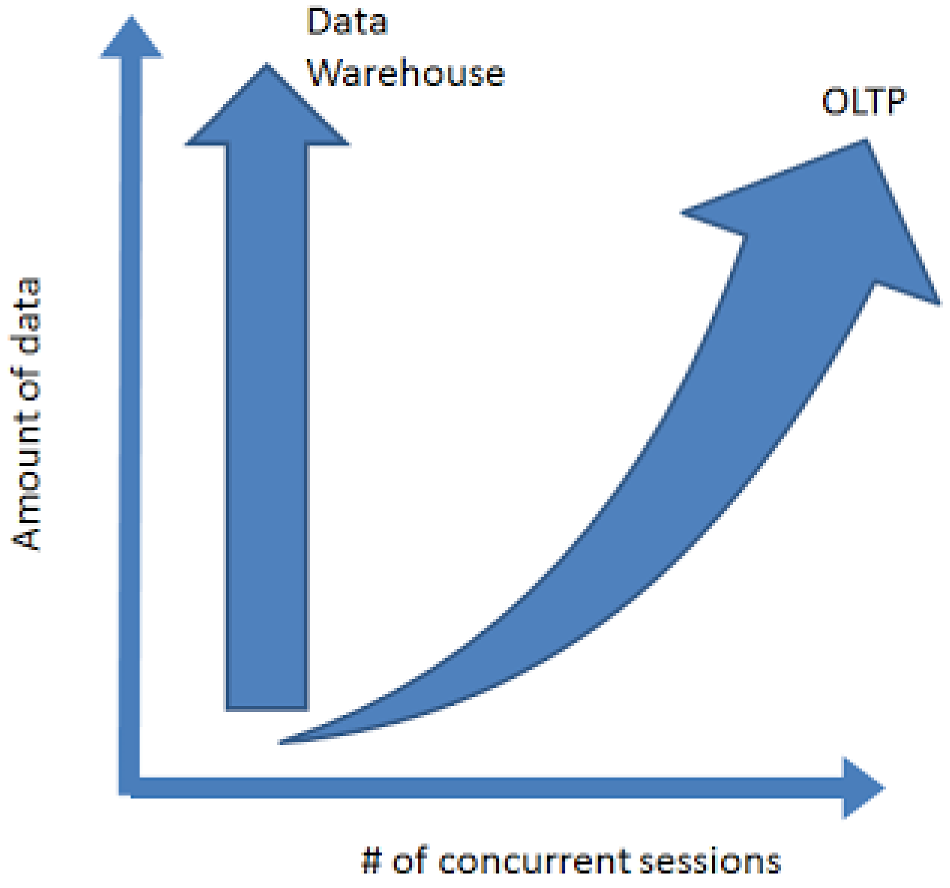
## SQL vs. NoSQL

## OLTP vs. OLAP

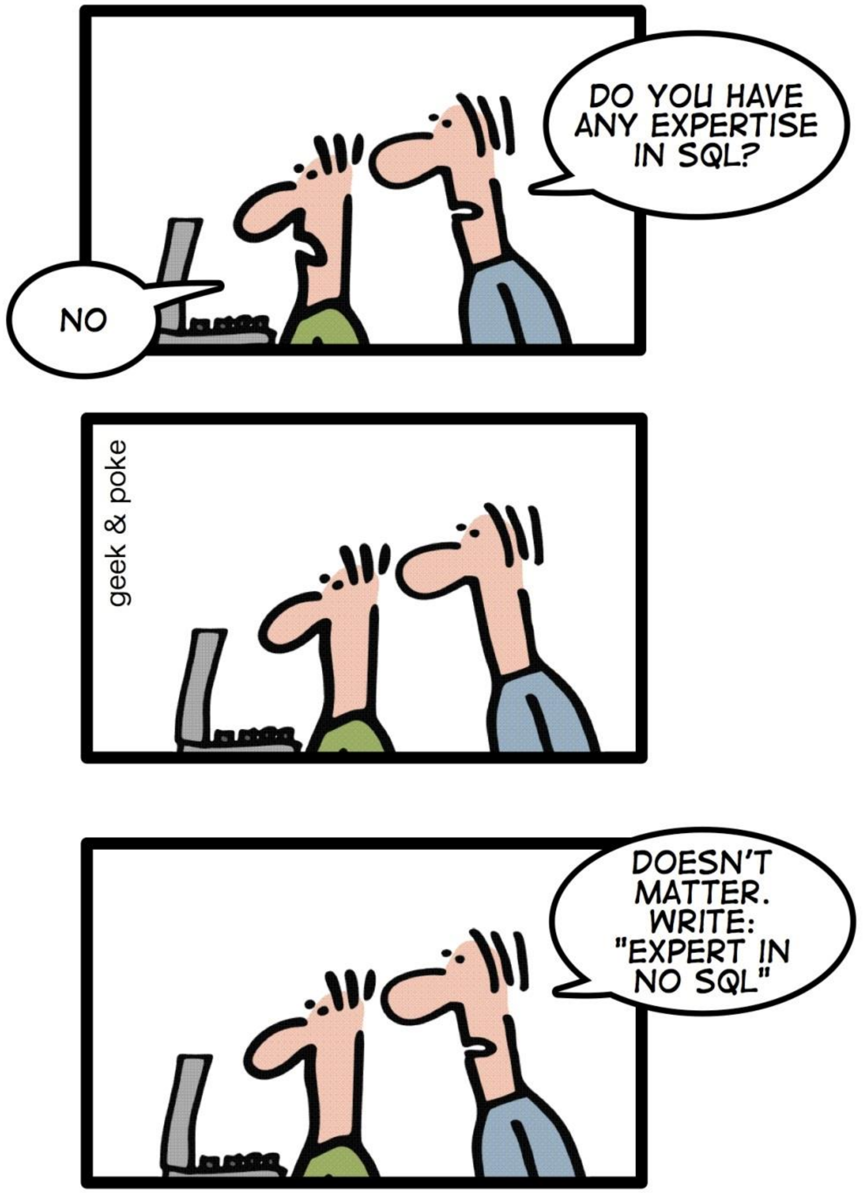
- We can divide IT systems into transactional (OLTP) and analytical (OLAP). In general we can assume that OLTP systems provide source data to data warehouses, whereas OLAP systems help to analyze it



## Challenges of Scale Differ



# HOW TO WRITE A CV


DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"

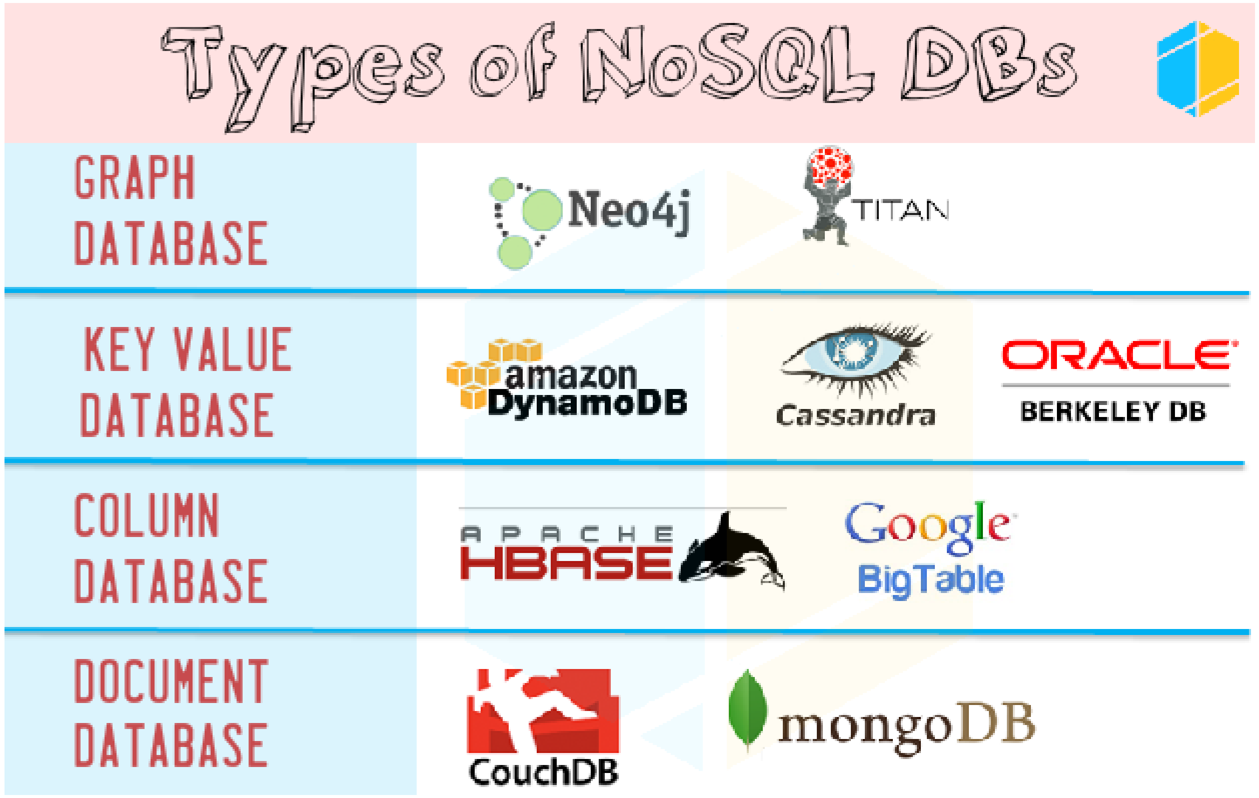
Leverage the NoSQL boom

## What is NoSQL?

- Not only SQL
- Data models other than strict tabular form
- Query models other than strict SQL

- Designed for un/semi/un/irregular structured data
- Designed to scale

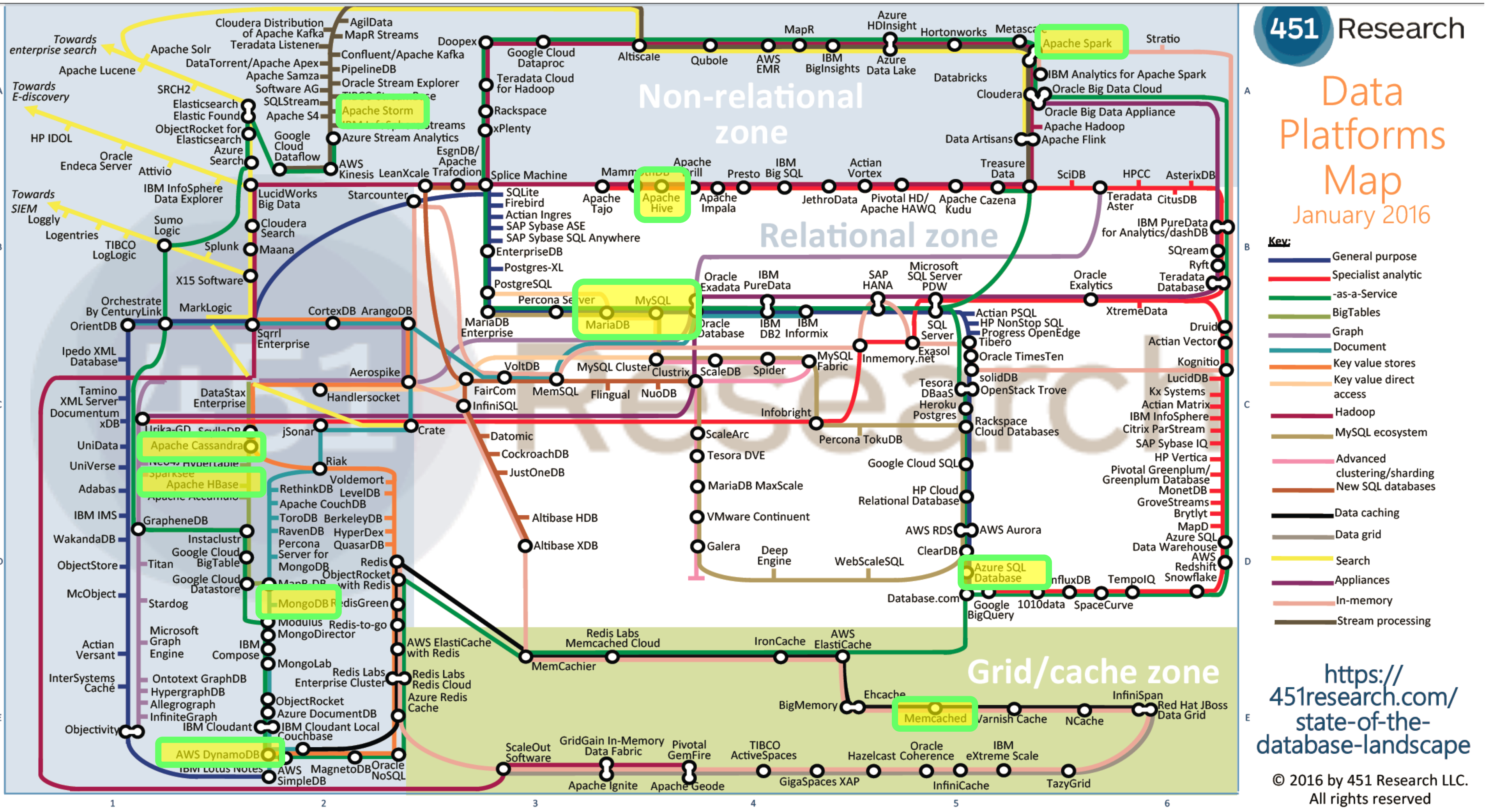
- Many different varieties!



# Types of NoSQL DBs

| | | | |
|---|---|---|---|
| GRAPH DATABASE | Neo4j | TITAN | |
| KEY VALUE DATABASE | amazon DynamoDB | Cassandra | ORACLE BERKELEY DB |
| COLUMN DATABASE | APACHE HBASE | Google BigTable | |
| DOCUMENT DATABASE | CouchDB | mongoDB | |

http://2.bp.blogspot.com/-6ufQu8Te8Tc/VoDRa2HcDGI/AAAAAAAALTw/tlzKrrtji58/s1600/Types%2Bof%2BNoSQL%2BDatabase.png


451 Research
Data Platforms Map
January 2016
Key:
General purpose
Specialist analytic
-as-a-Service
BigTables
Graph
Document
Key value stores
Key value direct access
Hadoop
MySQL ecosystem
Advanced clustering/sharding
New SQL databases
Data caching
Data grid
Search
Appliances
In-memory
Stream processing
https://451research.com/state-of-the-database-landscape
© 2016 by 451 Research LLC. All rights reserved
Non-relational zone
Relational zone
Grid/cache zone
Towards enterprise search
Towards E-discovery
Towards SIEM
Apache Lucene
Apache Solr
HP IDOL
Oracle Endeca Server
Attivio
IBM InfoSphere Data Explorer
Loggly
Logentries
Sumo Logic
TIBCO LogLogic
Splunk
X15 Software
Orchestrate By CenturyLink
MarkLogic
OrientDB
SRCH2
Elasticsearch
Elastic Found
ObjectRocket for Elasticsearch
Azure Search
Cloudera Distribution of Apache Kafka
Teradata Listener
DataTorrent/Apache Apex
Apache Samza
Software AG
SQLStream
Apache S4
Apache Storm
Google Cloud Dataflow
AWS Kinesis
AgilData
MapR Streams
Confluent/Apache Kafka
PipelineDB
Oracle Stream Explorer
Azure Stream Analytics
LeanXcale
EsgnDB/Apache Trafodion
LucidWorks Big Data
Cloudera Search
Maana
Starcounter
Doopex
Google Cloud Dataproc
Teradata Cloud for Hadoop
Rackspace
xPlenty
Altiscale
Qubole
AWS EMR
MapR
IBM BigInsights
Azure HDInsight
Azure Data Lake
Hortonworks
Metascale
Apache Spark
Stratio
Databricks
Cloudera
IBM Analytics for Apache Spark
Oracle Big Data Cloud
Oracle Big Data Appliance
Apache Hadoop
Data Artisans
Apache Flink
Splice Machine
Apache Tajo
Apache Hive
Apache Drill
Apache Impala
Presto
IBM Big SQL
JethroData
Actian Vortex
Pivotal HD/Apache HAWQ
Apache Cazena Kudu
Treasure Data
SciDB
HPCC
AsterixDB
Teradata Aster
CitusDB
IBM PureData for Analytics/dashDB
SQream
Ryft
Teradata Database
SQLite
Firebird
Actian Ingres
SAP Sybase ASE
SAP Sybase SQL Anywhere
EnterpriseDB
Postgres-XL
PostgreSQL
Percona Server
MySQL
MariaDB
MariaDB Enterprise
Oracle Exadata
IBM PureData
SAP HANA
Microsoft SQL Server PDW
Oracle Exalytics
Oracle Database
IBM DB2
IBM Informix
SQL Server
Actian PSQL
HP NonStop SQL
Progress OpenEdge
Tibero
Oracle TimesTen
solidDB
OpenStack Trove
Tesora DBaaS
Heroku Postgres
Rackspace Cloud Databases
XtremeData
Druid
Actian Vector
Kognitio
LucidDB
Kx Systems
Actian Matrix
IBM InfoSphere
Citrix ParStream
SAP Sybase IQ
HP Vertica
Pivotal Greenplum/Greenplum Database
MonetDB
GroveStreams
Brytlyt
MapD
Azure SQL Data Warehouse
AWS Redshift
Snowflake
Exasol
Inmemory.net
MySQL Fabric
Infobright
Percona TokuDB
Google Cloud SQL
HP Cloud Relational Database
AWS RDS
AWS Aurora
ClearDB
Azure SQL Database
InfluxDB
TempoIQ
Database.com
Google BigQuery
1010data
SpaceCurve
VoltDB
MySQL Cluster
Clustrix
ScaleDB
Spider
FairCom
MemSQL
Flingual
NuoDB
InfiniSQL
Datomic
CockroachDB
JustOneDB
Altibase HDB
Altibase XDB
ScaleArc
Tesora DVE
MariaDB MaxScale
VMware Continuent
Galera
Deep Engine
WebScaleSQL
Ipedo XML Database
Tamino XML Server
Documentum xDB
UniData
UniVerse
Adabas
IBM IMS
WakandaDB
ObjectStore
McObject
Actian Versant
InterSystems Caché
Objectivity
CortexDB
ArangoDB
Sqrrl Enterprise
Aerospike
DataStax Enterprise
Handlersocket
Urika-GD
ScyllaDB
Apache Cassandra
NCC49 Hypertable
Sparksee
Apache HBase
Apache Accumulo
jSonar
Crate
Riak
Voldemort
RethinkDB
LevelDB
Apache CouchDB
ToroDB
BerkeleyDB
RavenDB
HyperDex
Percona Server for MongoDB
QuasarDB
GrapheneDB
Instaclustr
Google Cloud BigTable
Titan
Google Cloud Datastore
Redis
ObjectRocket with Redis
MapR-DB
MongoDB
RedisGreen
Stardog
Modulus
MongoDirector
Redis-to-go
Microsoft Graph Engine
IBM Compose
AWS ElastiCache with Redis
MongoLab
Redis Labs Enterprise Cluster
Redis Labs Redis Cloud
Azure Redis Cache
Ontotext GraphDB
HypergraphDB
Allegrograph
InfiniteGraph
ObjectRocket
Azure DocumentDB
IBM Cloudant
IBM Cloudant Local
Couchbase
AWS DynamoDB
IBM Lotus Notes
AWS SimpleDB
MagnetoDB
Oracle NoSQL
Redis Labs Memcached Cloud
MemCachier
IronCache
AWS ElastiCache
BigMemory
Ehcache
Memcached
Varnish Cache
NCache
InfiniSpan
Red Hat JBoss Data Grid
ScaleOut Software
GridGain In-Memory Data Fabric
Apache Ignite
Pivotal GemFire
Apache Geode
TIBCO ActiveSpaces
GigaSpaces XAP
Hazelcast
Oracle Coherence
InfiniCache
IBM eXtreme Scale
TazyGrid

## ACID & BASE

## SQL Characteristics

- Data stored in columns and tables
- Relationships represented by data
- Declarative Languages
  - Data Manipulation Language
  - Data Definition Language
- Transactions
- Abstraction from physical layer
- ACID!

## ACID

- A model for correct behaviour of databases
  - *Name was coined (no surprise) in California in 60's*
- **Atomicity**: even if "transactions" have multiple operations, does them to completion (commit) or rolls back so that they leave no effect (abort)
- **Consistency**: A transaction that runs on a correct database leaves it in a correct ("consistent") state
- **Isolation**: It looks as if each transaction ran all by itself. Basically says "we'll hide any concurrency"
- **Durability**: Once a transaction commits, updates can't be lost or rolled back

## ACID eases development

- No need to worry about a transaction leaving some sort of partial state
  - For example, showing Tony as retired and yet leaving some customer accounts with him as the account rep
- Transaction can’t glimpse a partially completed state of some concurrent transaction
  - Eliminates worry about transient database inconsistency that might cause a transaction to crash
- Serial & Serializable Execution
  - Offers concurrency while hiding side-effects
- But costs are not small
  - $O(n^2)..O(n^5)$ for replicated ACID, *n* is replica set size

*Jim Gray, Pat Helland, Patrick E. O'Neil, Dennis Shasha: The Dangers of Replication and a Solution. SIGMOD 1996: 173-182*

## BASE

- Basically Available Soft-State Services with Eventual Consistency
  - Methodology for transforming transactional application into more concurrent & less rigid
  - Guide programmers to a cloud solution that performs much better
- Doesn't guarantee ACID properties
  - Uses the CAP Theorem

*BASE: An ACID Alternative, DAN PRITCHETT, May/June 2008 ACM QUEUE*

## BASE

- **Basically Available**
- *Goal is to promote rapid responses.*
- Partitioning faults are rare in data centers
  - Crashes force isolated machines to reboot
- Need rapid responses even when some replicas on critical path can't be contacted
  - Fast response even if some replicas are slow or crashed

## BASE

- **Soft State Service**
- Runs in first tier. *Can't store permanent data.*
- Restarts in a "clean" state after a crash
- To remember data:
  - Replicate it in memory in enough copies to never lose all in any crash
  - Pass it to some other service that keeps "hard state"

## BASE

- **Eventual Consistency**
- OK to send "optimistic" answers to external client
  - Send reply to user before finishing the operation
- Can use cached data (without staleness check)
- Can guess the outcome of an update
- Can skip locks, hoping no conflicts happen
- *Later, if needed, correct any inconsistencies in an offline cleanup activity*

- Developer ends up thinking hard and working hard!

## CAP Theorem

## Eric Brewer's CAP theorem

- In a famous 2000 keynote talk at ACM PODC, Eric Brewer proposed that *"you can have just two from Consistency, Availability and Partition Tolerance"*
  - He argues that data centers need very snappy response, hence availability is paramount
  - And they should be responsive even if a transient fault makes it hard to reach some service.
  - So they should use cached data to respond faster even if the cache can't be validated and might be stale!
- Conclusion: **weaken consistency for faster response**

CAP Twelve Years Later: How the "Rules" Have Changed, Eric Brewer, *IEEE Computer*, FEBRUARY 2012

# CAP theorem

- A proof of CAP was later introduced by MIT's Seth Gilbert and Nancy Lynch
  - Suppose a data center service is active in two parts of the country with a network link between them
  - We temporarily cut the link ("partitioning" the network)
  - And present the service with conflicting requests
- The replicas can't talk to each other so can't sense the conflict
- If they respond at this point, inconsistency arises

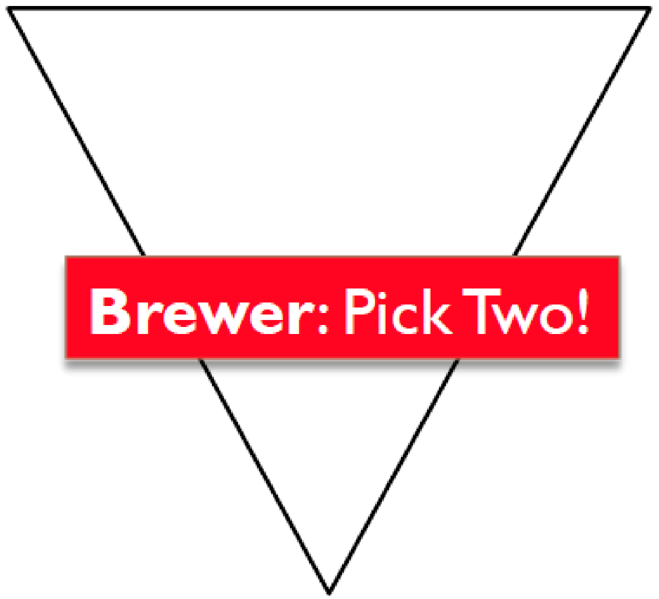
Perspectives on the CAP Theorem,
Seth Gilbert & Nancy A. Lynch, *IEEE Computer*, FEBRUARY 2012

**Atomic/Linearizable Consistency**

*Exist a total order of all Operations such that each operation looks as if it were completed at a single instant*

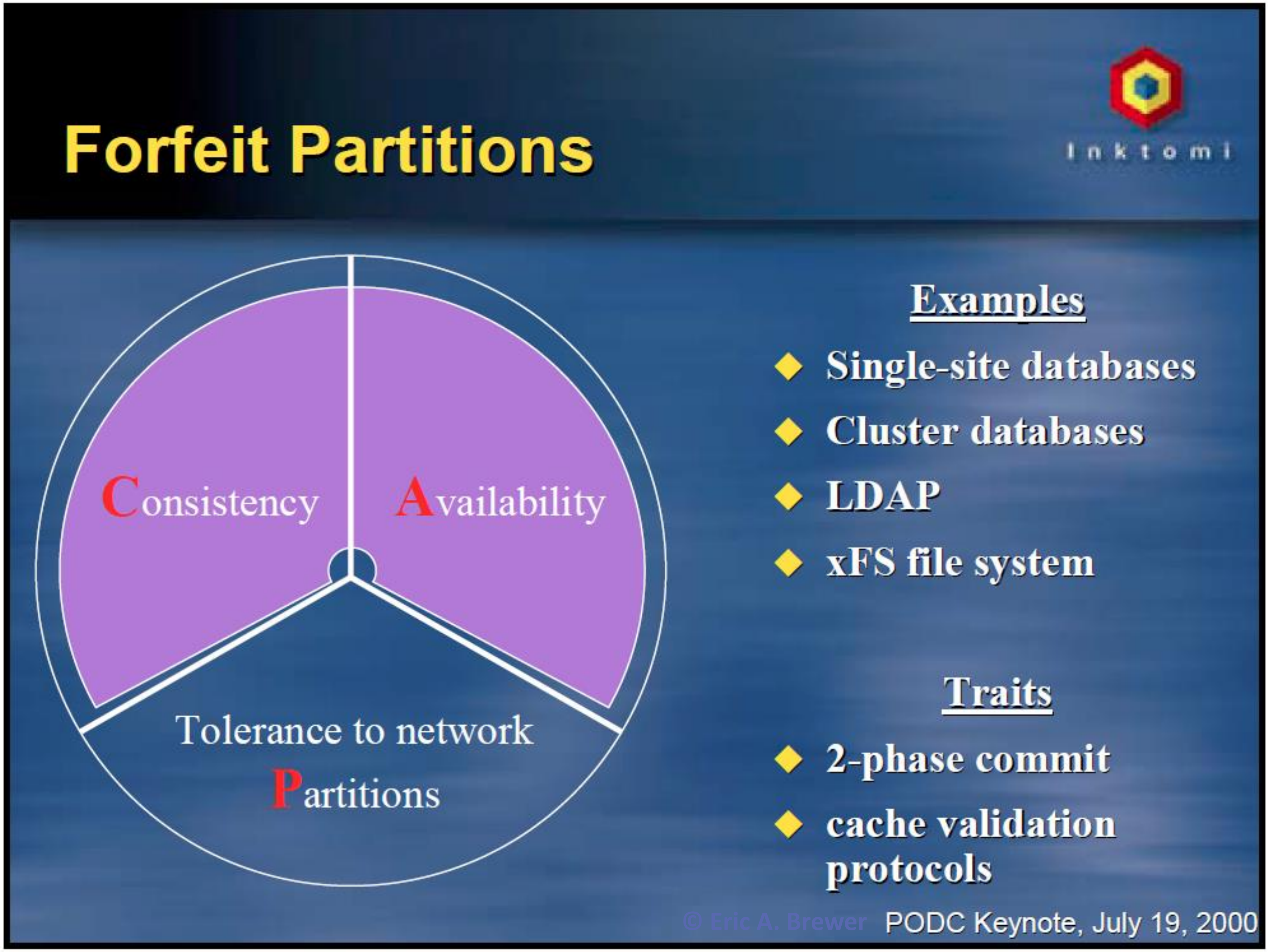
**Availability**

*Every request received by a non-failing node must result in a response*



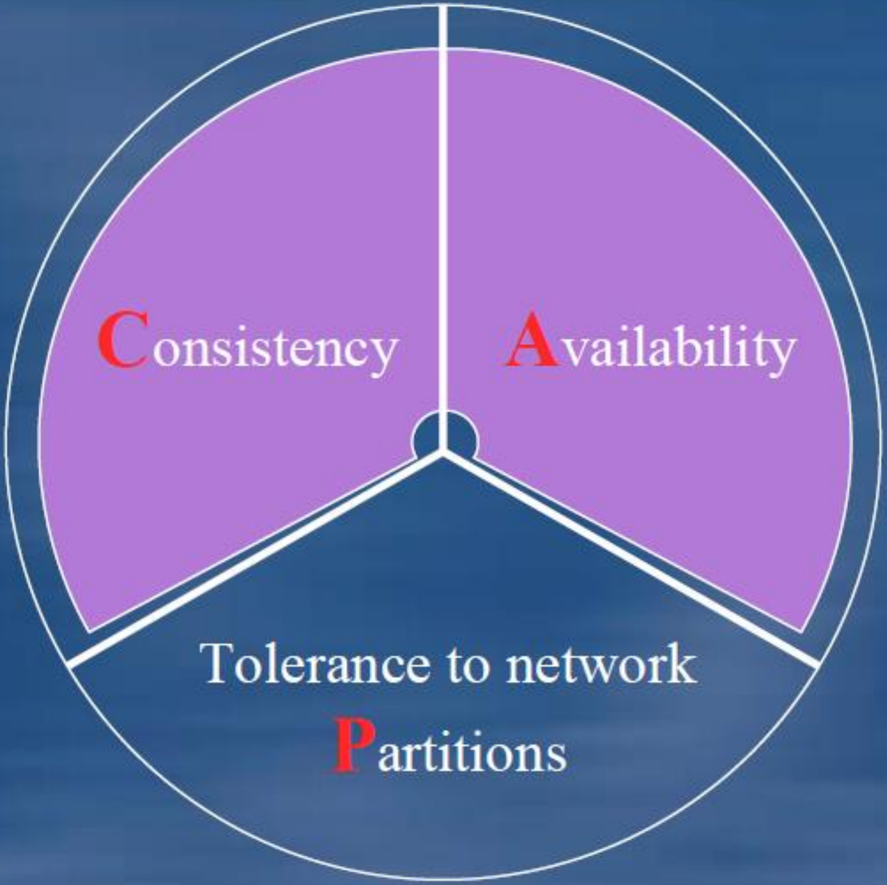
**Partition-tolerance**

*No set of failures less than total network failure Is allowed to cause the system to response incorrectly*



## Forfeit Partitions

Inktomi

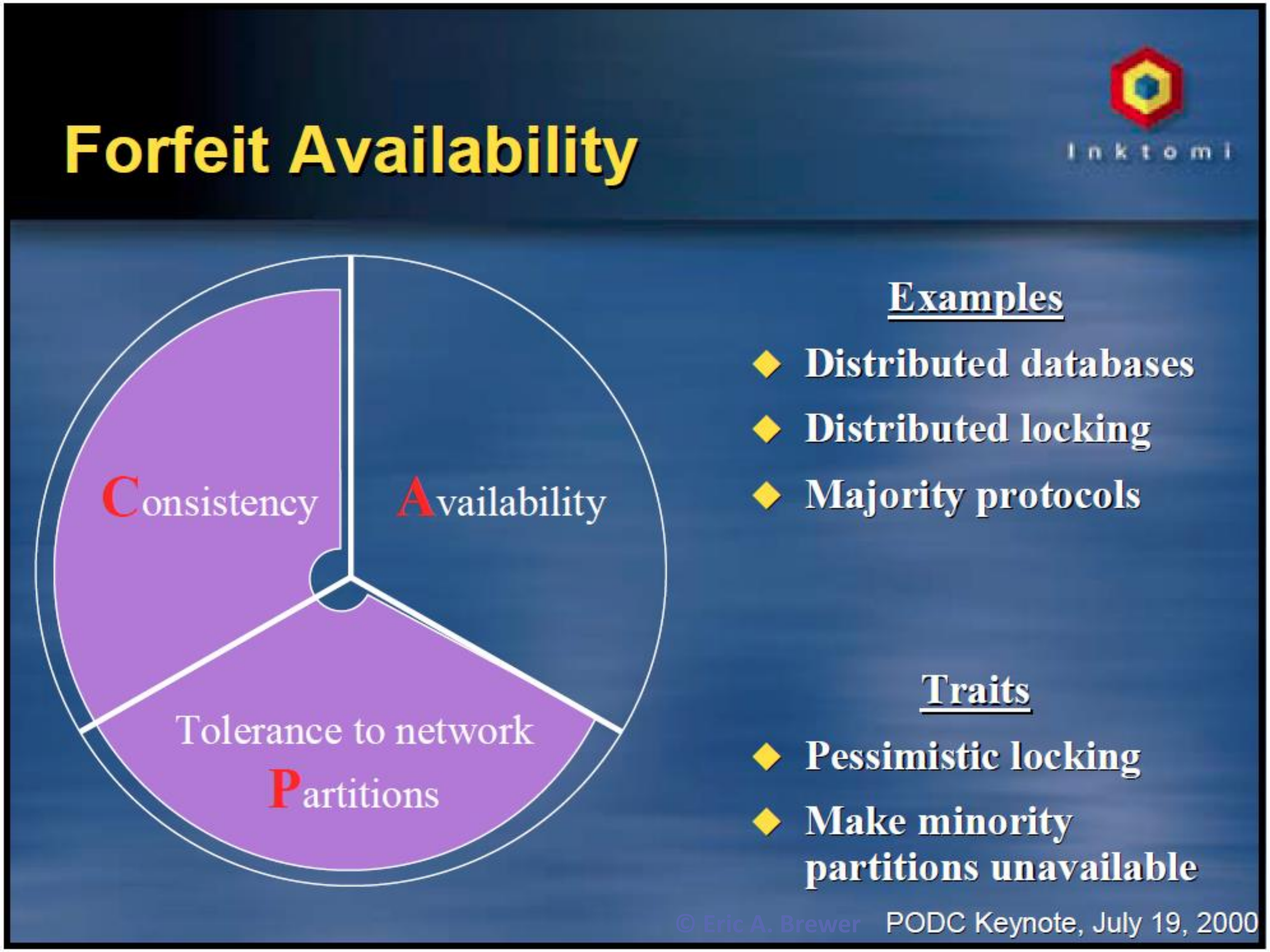


**Examples**

- Single-site databases
- Cluster databases
- LDAP
- xFS file system

**Traits**

- 2-phase commit
- cache validation protocols



## Forfeit Availability

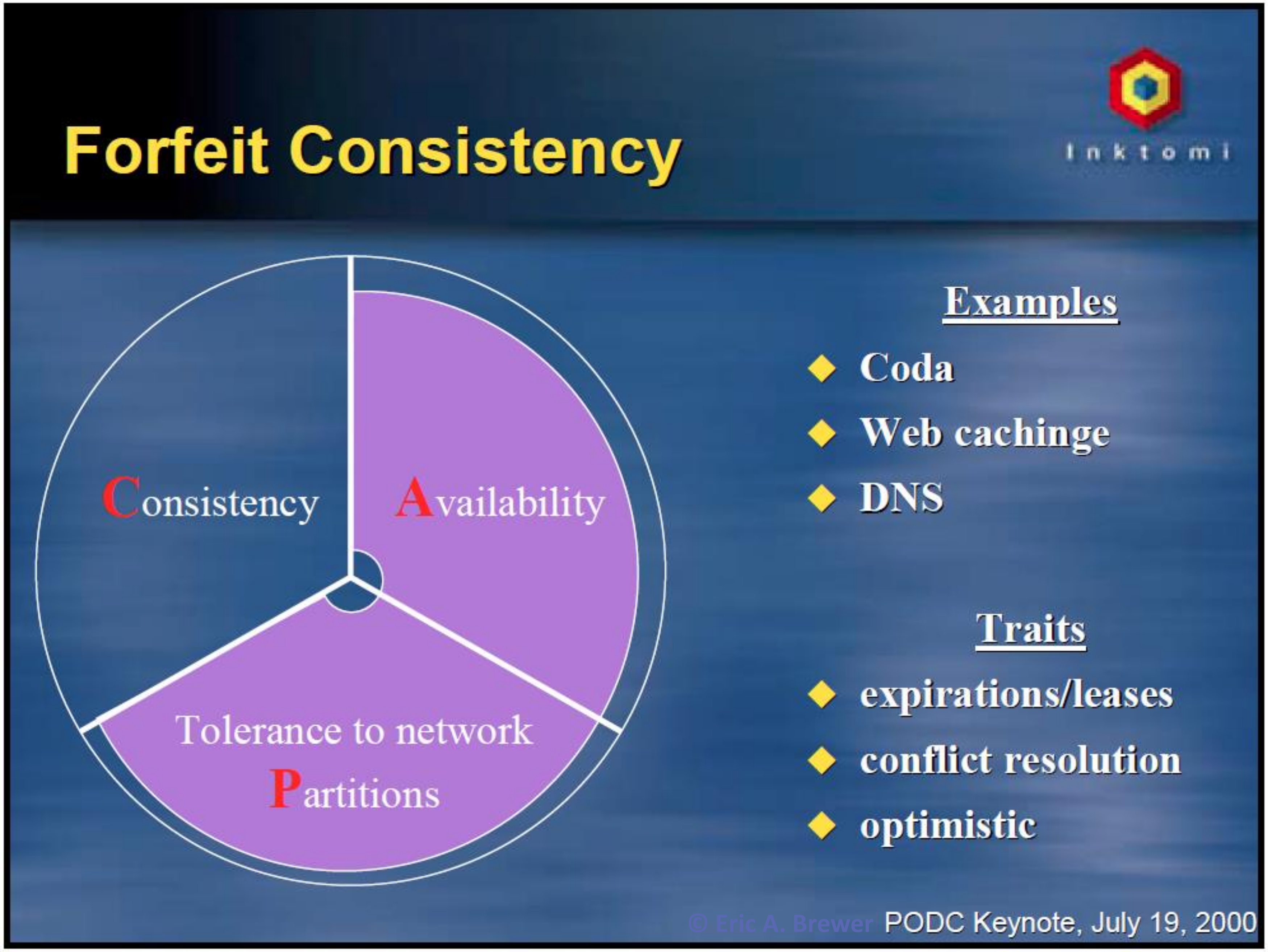
Consistency
Availability
Tolerance to network Partitions

**Examples**

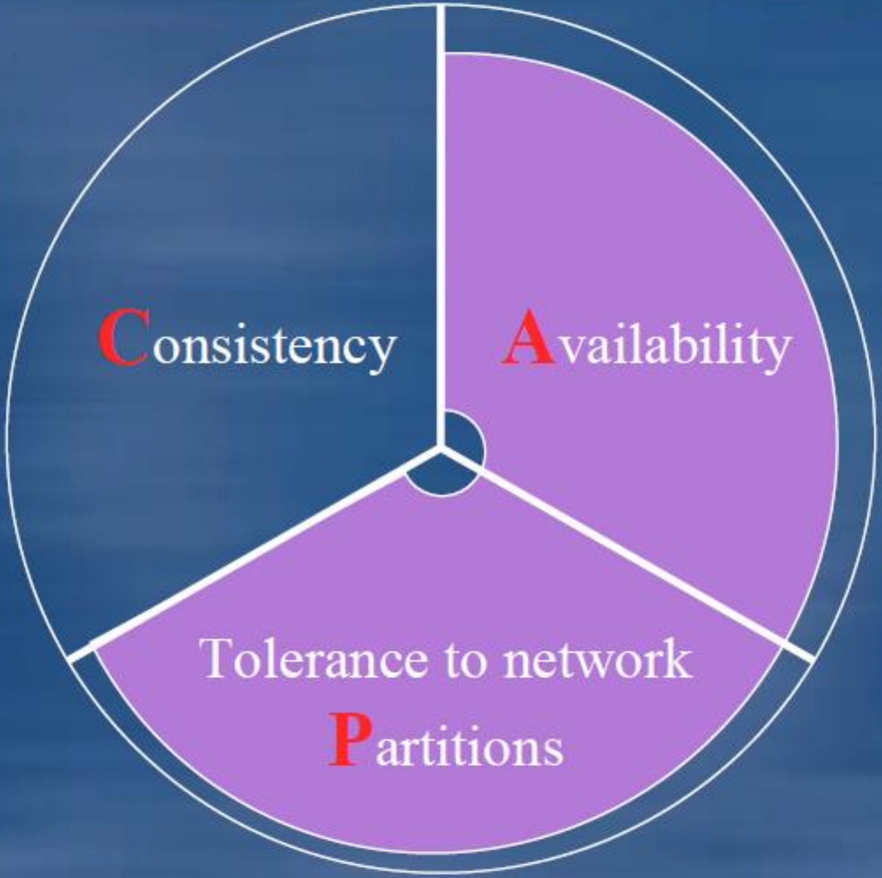
- Distributed databases
- Distributed locking
- Majority protocols

**Traits**

- Pessimistic locking
- Make minority partitions unavailable



## Forfeit Consistency



**Examples**
- Coda
- Web cachinge
- DNS

**Traits**
- expirations/leases
- conflict resolution
- optimistic

© Eric A. Brewer PODC Keynote, July 19, 2000

## Is inconsistency a bad thing?

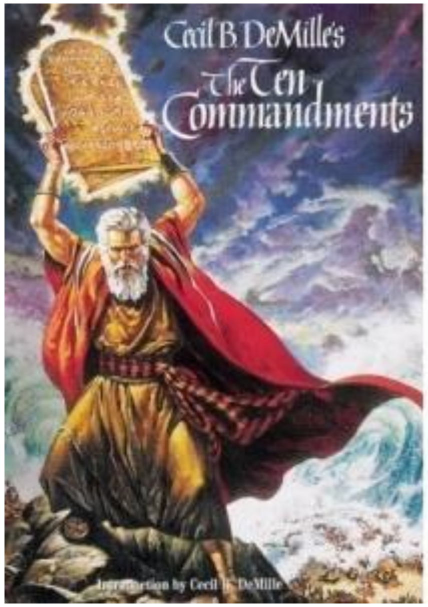
- How much consistency is really needed in the first tier of the cloud?
  - *Think about YouTube videos. Would consistency be an issue here?*
  - *What about the Amazon "number of units available" counters. Will people notice if those are a bit off?*
- **Puzzle**: Can you come up with a general policy for knowing how much consistency a given thing needs?

# eBay's Five Commandments

- As described by Randy Shoup at LADIS 2008

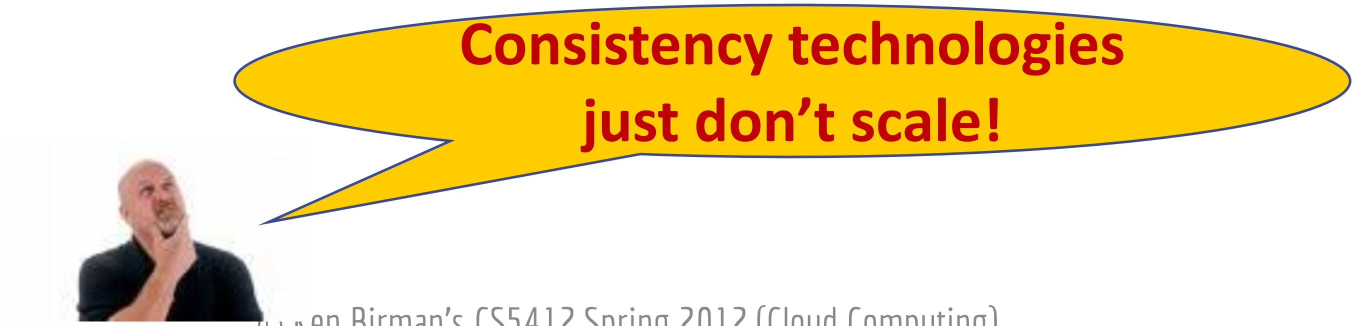
*Thou shalt…*

1. Partition Everything
2. Use Asynchrony Everywhere
3. Automate Everything
4. Remember: Everything Fails
5. Embrace Inconsistency



## Vogels at the Helm

- Werner Vogels is CTO at Amazon.com…
- He was involved in building a new shopping cart service
  - The old one used strong consistency for replicated data
  - New version was build over a DHT, like Chord, and has weak consistency with eventual convergence
- This weakens guarantees… but
  - ***Speed matters more than correctness***



## What does "consistency" mean?

- We need to pin this basic issue down!
- As used in CAP, consistency is about two things
  1. First, that *updates* to the *same data item* are applied in *some agreed-upon order*
  2. Second, that once an *update is acknowledged* to an external user, it *won't be forgotten*
- Not all systems need both properties

# Apache Dynamo DB

Key Value Store

*Cassandra*

# Amazon's Dynamo DB

- Key-Value Store
  - Simple **`Get()`** & **`Put()`** operations on objects with unique ID. No queries.
- Highly Available
  - Even the slightest outage has significant financial consequences
- Service Level Agreements
  - Guaranteeing response in 300ms for 99.9% of requests at a peak load of 500 req/sec

*Dynamo: Amazon's Highly Available Key-value Store, Giuseppe DeCandia, et al, SOSP, 2007*

## Design Choices

- Sacrifice strong consistency for availability
  - "always writeable". No updates are rejected.
  - Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".
- Incremental scalability & decentralization
  - Symmetry of responsibility
  - Heterogeneity in capacity
- All nodes are trusted

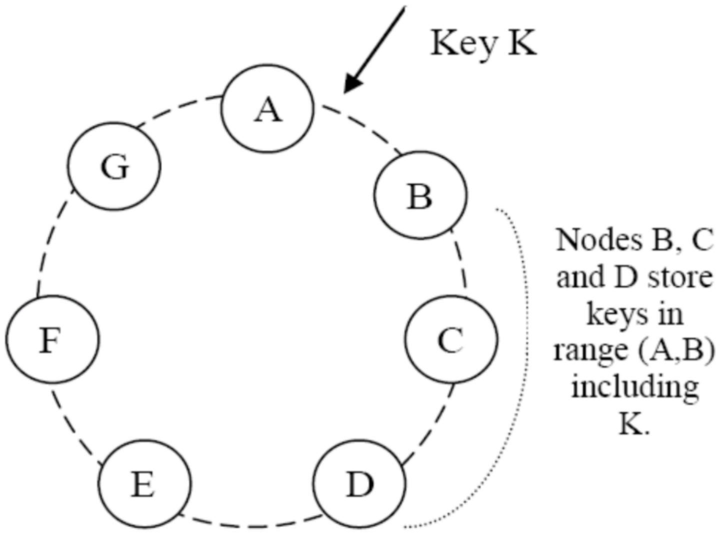
# Techniques

| Problem | Technique | Advantage |
|---|---|---|
| **Partitioning** | Consistent Hashing | Incremental Scalability |
| **High Availability for writes** | Vector clocks with reconciliation during reads | Version size is decoupled from update rates. |
| **Handling temporary failures** | Sloppy Quorum and hinted handoff | Provides high availability and durability guarantee when some of the replicas are not available. |
| **Recovering from permanent failures** | Anti-entropy using Merkle trees | Synchronizes divergent replicas in the background. |
| **Membership and failure detection** | Gossip-based membership protocol and failure detection. | Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information. |

*From external sources*

# Partitioning

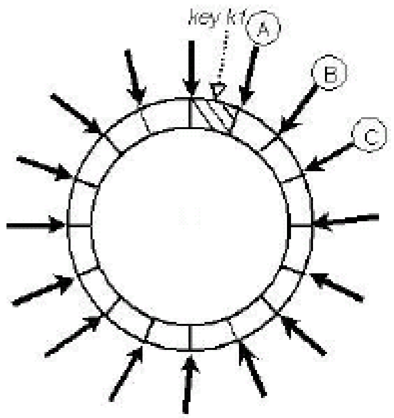
- Consistent hashing
- Output range of hash func. on key is a fixed "ring"
- Virtual node is responsible for a range of hash values (tokens)
  - Hash value for the key maps to a virtual node
- Each physical node responsible for multiple virtual nodes
  - Allows nodes to arrive and leave without having to change keys present in virtual nodes



*From external sources*

# Partitioning and placement of key

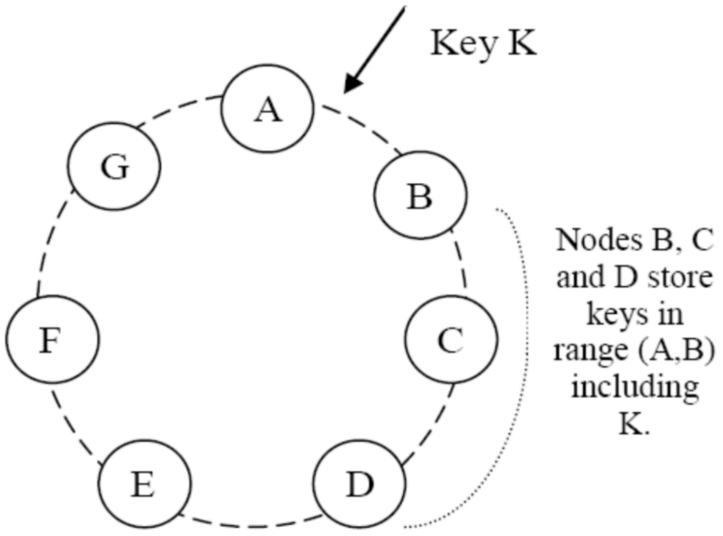
- Divide the hash space into Q equally sized partitions…virtual node or token
- Each physical node assigned Q/S tokens where S is the number of nodes in the system.
  - Can also assign variable tokens to physical node based on machine size
- Adapt to capacity of physical nodes
- Incrementally add/remove physical nodes
  - When a node leaves the system, its tokens (virtual nodes) are randomly & uniformly distributed to the remaining nodes to load balance
  - When a node joins the system it uniformaly "steals" tokens from nodes in the system to load balance



*From external sources*

# Replication

- Each data item is replicated at N hosts.
  - "*preference list*": The list of nodes responsible for storing a particular key.
- *Coordinator node* (from hashing) stores first copy
  - Next copy stored in subsequent virtual nodes
  - Skip virtual nodes present on same physical node
- Gossip protocol
  - Propagates changes among nodes
  - Eventually consistent view of membership, mapping from tokens to nodes



*D stores (A, B], (B, C], (C, D]*

*From external sources*

## Key Value Operations

- Add and update items both use **put(key, value)** operation
- **get(key)** returns the value
- Any node may receive the request
- Forwarded to the coordinator node for response
- **put()** may return to its client before the update is applied at all replicas
  - May leave replicas in inconsistent state
- **get()** may return many versions of same object

## Sloppy Quorum

- Writes are successful if 'w' replicas can be updated (w<N)
  - Coordinator forwards requests to all N replicas, and returns when 'w' respond
- Reads return all 'r' replica values (r<N)
  - Coordinator sends requests to all N replicas, and returns when 'r' respond
  - Clients need to decide how to use these copies
- Reads & writes dictated by *slowest replica*
  - Set **r+w > N**

*From external sources*

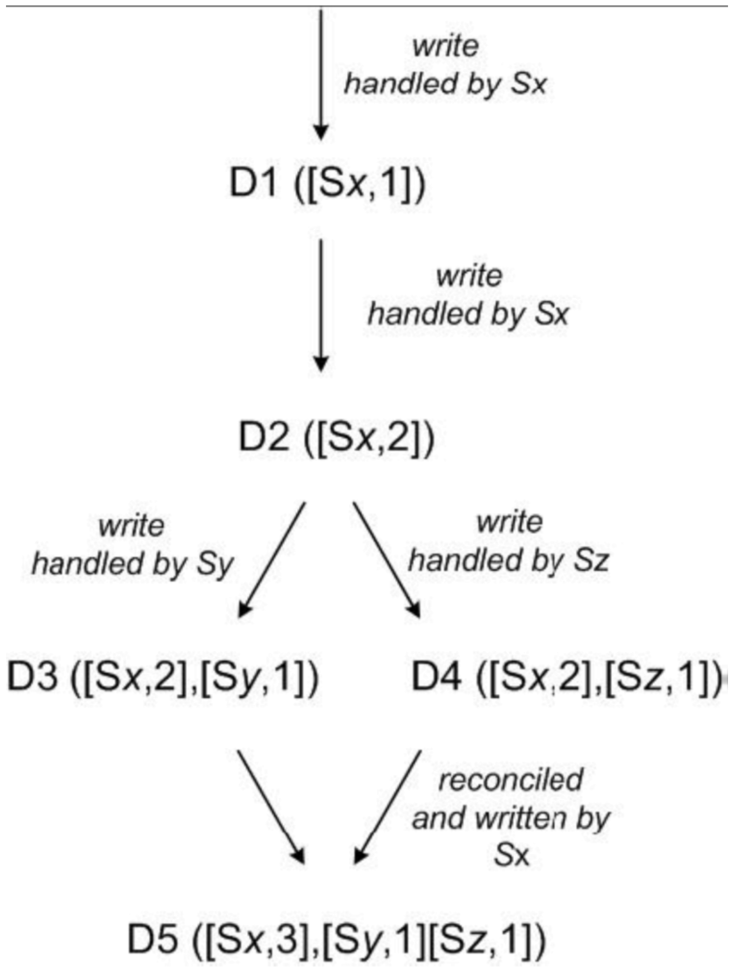
## Data Versioning & Consistency

- put() is treated as *append* of the updated value
  - Immutable append to a *particular version* of the object
  - Multiple versions can coexist…but system will not internally "resolve" them
- Challenge
  - *Distinct version sub-histories need to be reconciled.*
- Solution
  - *Uses vector clocks to capture causality between different versions of the same object.*

*From external sources*

# Consistency with Vector Clocks

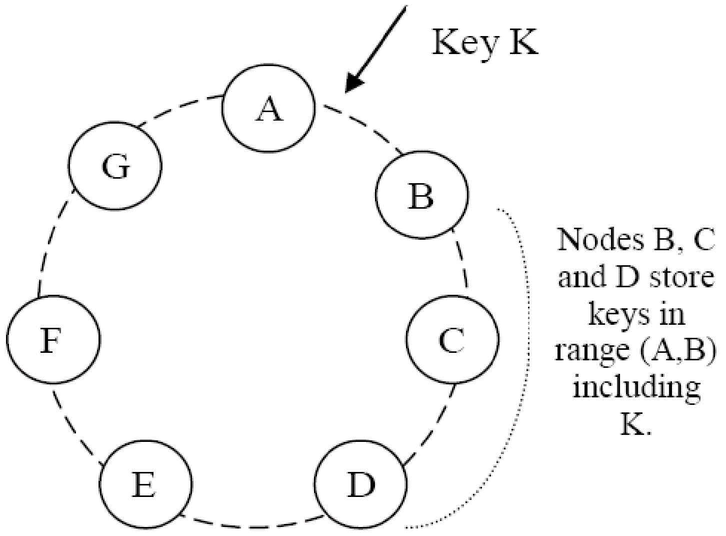
- Vector clock: (node, counter) pair
  - Every version of every object is associated with one vector clock.
  - If the counters on the first object's clock are <= all nodes in the second clock, then the first is an ancestor of the second and can be forgotten.
  - i.e. first object happened before second object
- If get() has multiple replica versions, return causally "unrelated" versions
  - *i.e. remove partial ordered & only return causally unordered versions for reconciliation*
- Client writes the reconciled version back
  - e.g. Sx resolves D3 and D4 into D5



From external sources

## Hinted handoff

- Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.
- D is hinted that the replica is belong to A and it will deliver to A when A is recovered.
- Again: “always writeable”



*From external sources*

## Replica synchronization

- Merkle tree:
  - a hash tree where leaves are hashes of the values of individual keys.
  - Parent nodes higher in the tree are hashes of their respective children.

*From external sources*

## Replica synchronization

- Advantage of Merkle tree:
  - Each branch of the tree can be checked independently without requiring nodes to download the entire tree.
  - Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas.

*From external sources*

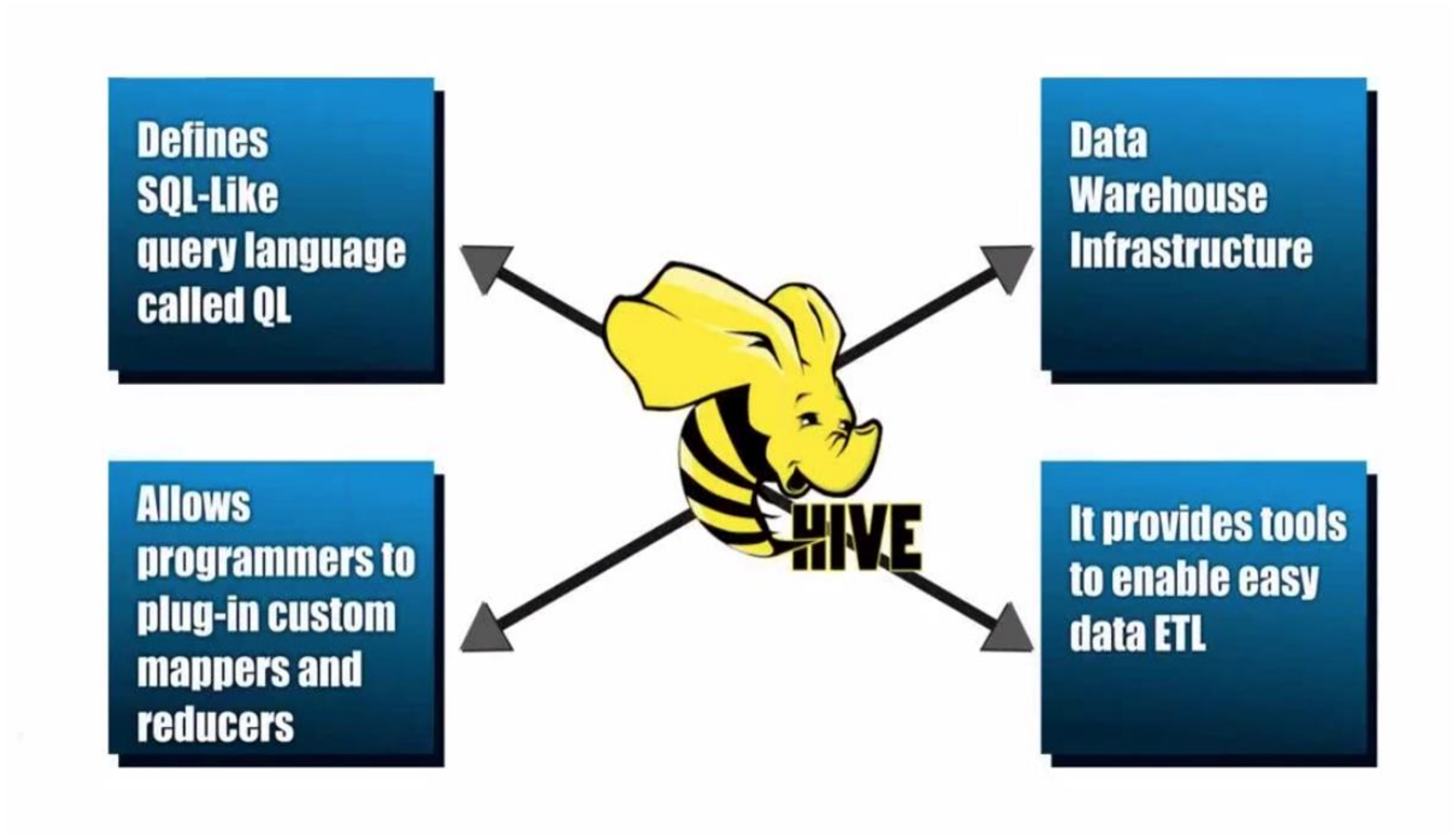
## Self study: Cassandra

- Intro to Apache Cassandra, Philip Thompson, DataStax
  - https://www.youtube.com/watch?v=oawc4doC76U
  - Q&A next Tuesday

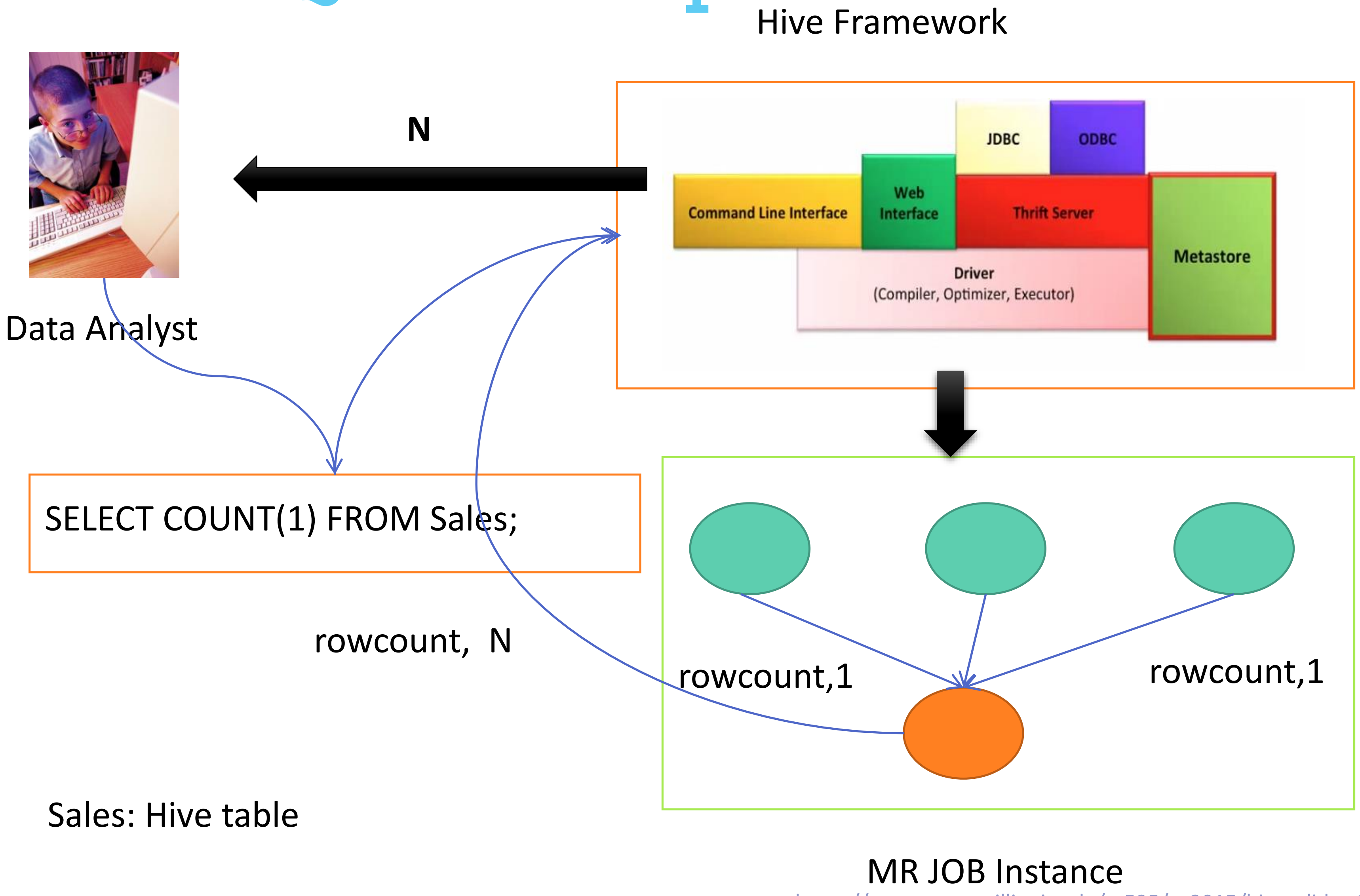
## Hive

SQL Data Warehouse

## Hive Key Principles


Defines SQL-Like query language called QL
Data Warehouse Infrastructure
HIVE
Allows programmers to plug-in custom mappers and reducers
It provides tools to enable easy data ETL

# HiveQL to MapReduce



```
SELECT COUNT(1) FROM Sales;
```

## What Is Hive?

- Developed by Facebook and a top-level Apache project
- A data warehousing infrastructure based on Hadoop
- Immediately makes data on a cluster available to non-Java programmers via SQL like queries
- Built on HiveQL (HQL), a SQL-like query language
- Interprets HiveQL and generates MapReduce jobs that run on the cluster
- Enables easy data summarization, ad-hoc reporting and querying, and analysis of large volumes of data

## What Hive Is Not

- Hive, like Hadoop, is designed for batch processing of large datasets
- Not an OLTP or real-time system
- Latency and throughput are both high compared to a traditional RDBMS
  - Even when dealing with relatively small data (<100 MB)

## Data Hierarchy

- Hive is organised hierarchically into:
  - **Databases**: namespaces that separate tables and other objects
  - **Tables**: homogeneous units of data with the same schema
    - Analogous to tables in an RDBMS
  - **Partitions**: determine how the data is stored
    - Allow efficient access to subsets of the data
  - **Buckets/clusters**
    - For sub-sampling within a partition
    - Join optimization

## Tables

- Analogous to relational tables
- Each table has a corresponding directory in HDFS
- Data serialized and stored as files within that directory
- Hive has default serialization built in which supports compression and lazy deserialization
- Users can specify custom serialization – deserialization schemes (SerDe's)

## Partitions

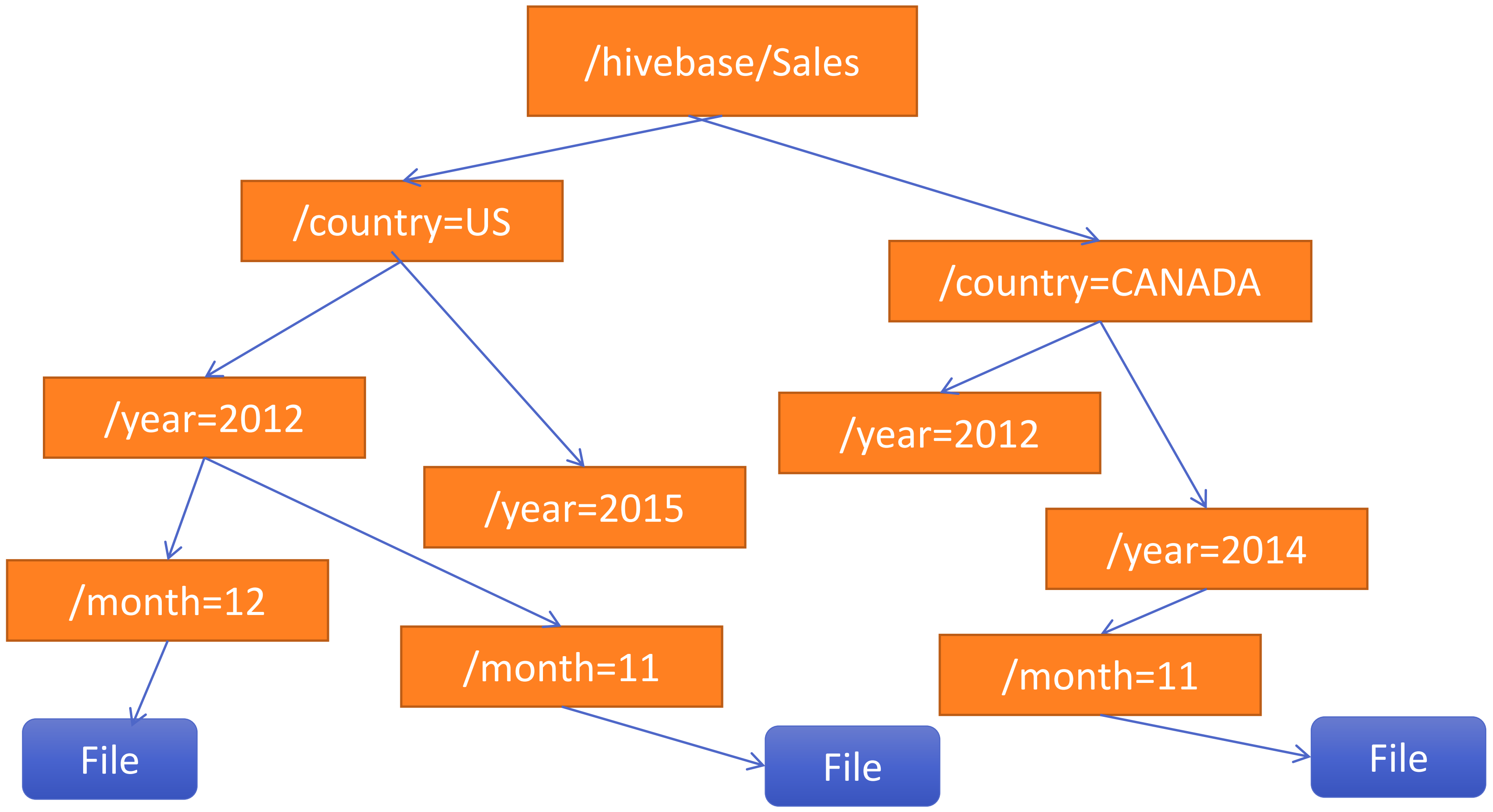
- Each table can be broken into partitions
- Partitions determine distribution of data within subdirectories
- Example -

CREATE_TABLE Sales (sale_id INT, amount FLOAT)

PARTITIONED BY (country STRING, year INT, month INT)

- So each partition will be split out into different folders like
- Sales/country=US/year=2012/month=12

# Hierarchy of Hive Partitions



## Buckets

- Data in each partition divided into buckets
- Based on a hash function of the column
- H(column) mod NumBuckets = bucket number
- Each bucket is stored as a file in partition directory

## HiveQL

- HiveQL / HQL provides the basic SQL-like operations:
  - Select columns using SELECT
  - Filter rows using WHERE
  - JOIN between tables
  - Evaluate aggregates using GROUP BY
  - Store query results into another table
  - Download results to a local directory (i.e., export from HDFS)
  - Manage tables and queries with CREATE, DROP, and ALTER

## Primitive Data Types

| Type | Comments |
|---|---|
| TINYINT, SMALLINT, INT, BIGINT | 1, 2, 4 and 8-byte integers |
| BOOLEAN | TRUE/FALSE |
| FLOAT, DOUBLE | Single and double precision real numbers |
| STRING | Character string |
| TIMESTAMP | Unix-epoch offset *or* datetime string |
| DECIMAL | Arbitrary-precision decimal |
| BINARY | Opaque; ignore these bytes |

## Complex Data Types

| Type | Comments |
|---|---|
| STRUCT | A collection of elements<br>If S is of type STRUCT {a INT, b INT}:<br>S.a returns element a |
| MAP | Key-value tuple<br>If M is a map from 'group' to GID:<br>M['group'] returns value of GID |
| ARRAY | Indexed list<br>If A is an array of elements ['a','b','c']:<br>A[0] returns 'a' |

## HiveQL Limitations

- HQL only supports *equi-joins, outer joins, left semi-joins*
- Because it is only a shell for mapreduce, complex queries can be hard to optimise
- Missing large parts of full SQL specification:
  - HAVING clause in SELECT
  - Correlated sub-queries
  - Sub-queries outside FROM clauses
  - Updatable or materialized views
  - Stored procedures

## Hive Metastore

- Stores Hive metadata
- Default metastore database uses Apache Derby
- Various configurations:
  - **Embedded** (in-process metastore, in-process database)
    - Mainly for unit tests
  - **Local** (in-process metastore, out-of-process database)
    - Each Hive client connects to the metastore directly
  - **Remote** (out-of-process metastore, out-of-process database)
    - Each Hive client connects to a metastore server, which connects to the metadata database itself

## Hive Schemas

- Hive is schema-on-read
  - Schema is only enforced when the data is read (at query time)
  - Allows greater flexibility: same data can be read using multiple schemas
- Contrast with an RDBMS, which is schema-on-write
  - Schema is enforced when the data is loaded
  - Speeds up queries at the expense of load times

## Create Table Syntax

```
CREATE TABLE table_name
    (col1 data_type,
     col2 data_type,
     col3 data_type,
     col4 datatype )
   ROW FORMAT DELIMITED
   FIELDS TERMINATED BY ','
   STORED AS format_type;
```

## Simple Table

```
CREATE TABLE page_view
    (viewTime INT,
     userid BIGINT,
     page_url STRING,
     referrer_url STRING,
     ip STRING COMMENT 'IP Address of the User' )
   ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '\t'
   STORED AS TEXTFILE;
```

## More Complex Table

```
CREATE TABLE employees  (
    (name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
                  city:STRING,
                  state:STRING,
                  zip:INT>)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t'
  STORED AS TEXTFILE;
```

## External Table

```
CREATE EXTERNAL TABLE page_view_stg
    (viewTime INT,
     userid BIGINT,
     page_url STRING,
     referrer_url STRING,
     ip STRING COMMENT 'IP Address of the User')
   ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '\t'
   STORED AS TEXTFILE
   LOCATION '/user/staging/page_view';
```

## More About Tables

- CREATE TABLE
  - LOAD: file moved into Hive's data warehouse directory
  - DROP: both metadata and data deleted
- CREATE EXTERNAL TABLE
  - LOAD: no files moved
  - DROP: only metadata deleted
  - Use this when sharing with other Hadoop applications, or when you want to use multiple schemas on the same data

## Partitioning

- Can make some queries faster
- Divide data based on partition column
- Use PARTITION BY clause when creating table
- Use PARTITION clause when loading data
- SHOW PARTITIONS will show a table's partitions

## Bucketing

- Can speed up queries that involve sampling the data
  - Sampling works without bucketing, but Hive has to scan the entire dataset
- Use CLUSTERED BY when creating table
  - For sorted buckets, add SORTED BY
- To query a sample of your data, use TABLESAMPLE

# Browsing Tables And Partitions

| Command | Comments |
|---|---|
| `SHOW TABLES;` | Show all the tables in the database |
| `SHOW TABLES 'page.*';` | Show tables matching the specification ( uses regex syntax ) |
| `SHOW PARTITIONS page_view;` | Show the partitions of the page_view table |
| `DESCRIBE page_view;` | List columns of the table |
| `DESCRIBE EXTENDED page_view;` | More information on columns (useful only for debugging ) |
| `DESCRIBE page_view PARTITION (ds='2008-10-31');` | List information about a partition |

## Loading Data

- Use LOAD DATA to load data from a file or directory
  - Will read from HDFS unless LOCAL keyword is specified
  - Will append data unless OVERWRITE specified
  - PARTITION required if destination table is partitioned

```
LOAD DATA LOCAL INPATH '/tmp/pv_2008-06-
8_us.txt'
  OVERWRITE INTO TABLE page_view
  PARTITION (date='2008-06-08', country='US')
```

## Inserting Data

- Use INSERT to load data from a Hive query
  - Will append data unless OVERWRITE specified
  - PARTITION required if destination table is partitioned

```
FROM page_view_stg pvs
 INSERT OVERWRITE TABLE page_view
 PARTITION (dt='2008-06-08',
country='US')
  SELECT pvs.viewTime, pvs.userid,
         pvs.page_url, pvs.referrer_url
 WHERE pvs.country = 'US';
```

## Inserting Data

- Normally only one partition can be inserted into with a single INSERT
- A multi-insert lets you insert into multiple partitions

```
FROM page_view_stg pvs
INSERT OVERWRITE TABLE page_view
PARTITION  ( dt='2008-06-08', country='US‘ )
SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url WHERE
pvs.country = 'US'
INSERT OVERWRITE TABLE page_view
PARTITION ( dt='2008-06-08', country='CA' )
SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url WHERE
pvs.country = 'CA'
INSERT OVERWRITE TABLE page_view
PARTITION ( dt='2008-06-08', country='UK' )
SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url WHERE
pvs.country = 'UK';
```

## Inserting Data During Table Creation

- Use AS SELECT in the CREATE TABLE statement to populate a table as it is created

```
CREATE TABLE page_view AS
  SELECT pvs.viewTime, pvs.userid, pvs.page_url,
         pvs.referrer_url
  FROM page_view_stg pvs
  WHERE pvs.country = 'US';
```

# Loading And Inserting Data: Summary

| Use this | For this purpose |
|---|---|
| `LOAD` | Load data from a file or directory |
| `INSERT` | Load data from a query<br>• One partition at a time<br>• Use multiple INSERTs to insert into multiple partitions in the one query |
| `CREATE TABLE AS (CTAS)` | Insert data while creating a table |
| Add/modify external file | Load new data into external table |

## Sample Select Clauses

- Select from a single table

```
    SELECT *
          FROM sales
          WHERE amount > 10 AND
                          region = "US";
```

- Select from a partitioned table

```
    SELECT page_views.*
  FROM page_views
  WHERE page_views.date >= '2008-03-01' AND
      page_views.date <= '2008-03-31'
```

## Relational Operators

- ALL and DISTINCT
  - Specify whether duplicate rows should be returned
  - ALL is the default  (all matching rows are returned)
  - DISTINCT removes duplicate rows from the result set
- WHERE
  - Filters by expression
  - Does not support *IN, EXISTS* or *sub-queries* in the WHERE clause
- LIMIT
  - Indicates the number of rows to be returned

## Relational Operators

- GROUP BY
  - Group data by column values
  - Select statement can only include columns included in the GROUP BY clause
- ORDER BY / SORT BY
  - ORDER BY performs total ordering
    - Slow, poor performance
  - SORT BY performs partial ordering
    - Sorts output from each reducer

# Advanced Hive Operations

- JOIN
  - If only one column in each table is used in the join, then only one MapReduce job will run
    - This results in 1 MapReduce job:

      ```
      SELECT * FROM a JOIN b ON a.key = b.key JOIN c ON
      b.key = c.key
      ```

    - This results in 2 MapReduce jobs:

      ```
      SELECT * FROM a JOIN b ON a.key = b.key JOIN c ON
      b.key2 = c.key
      ```

  - If multiple tables are joined, put the biggest table last and the reducer will stream the last table, buffer the others

## Advanced Hive Operations

- JOIN
  - Do not specify join conditions in the WHERE clause
    - Hive does not know how to optimise such queries
    - Will compute a full Cartesian product before filtering it
- Join Example

```
SELECT
  a.ymd, a.price_close, b.price_close
FROM stocks a
JOIN stocks b ON a.ymd = b.ymd
WHERE a.symbol = 'AAPL' AND
      b.symbol = 'IBM' AND
      a.ymd > '2010-01-01';
```

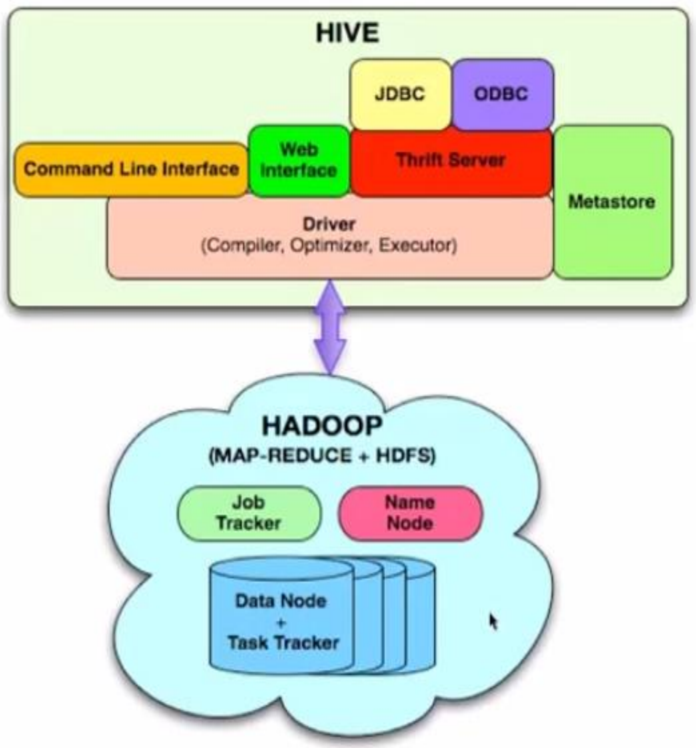
# Architecture

**Externel Interfaces**- CLI, WebUI, JDBC, ODBC programming interfaces

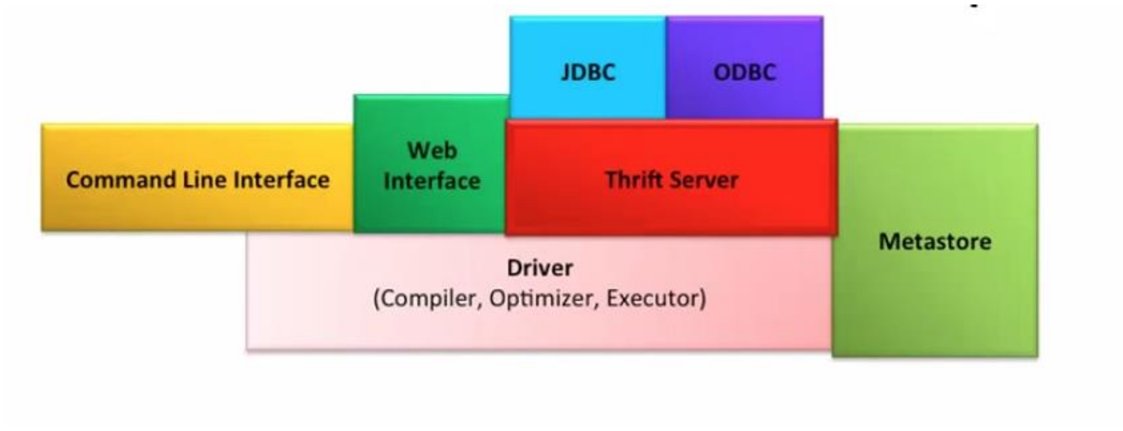
**Thrift Server** – Cross Language service framework .

**Metastore** - Meta data about the Hive tables, partitions

**Driver** - Brain of Hive! Compiler, Optimizer and Execution engine

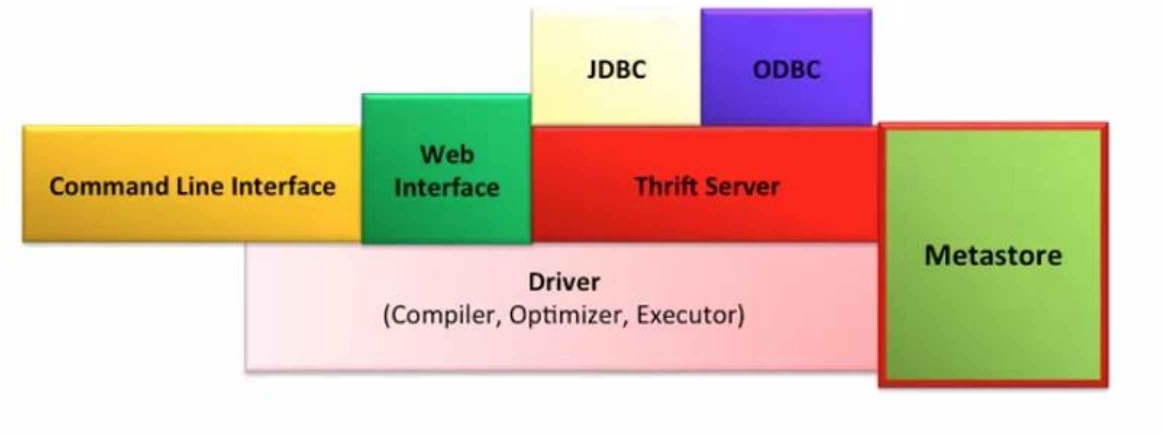


# Hive Thrift Server



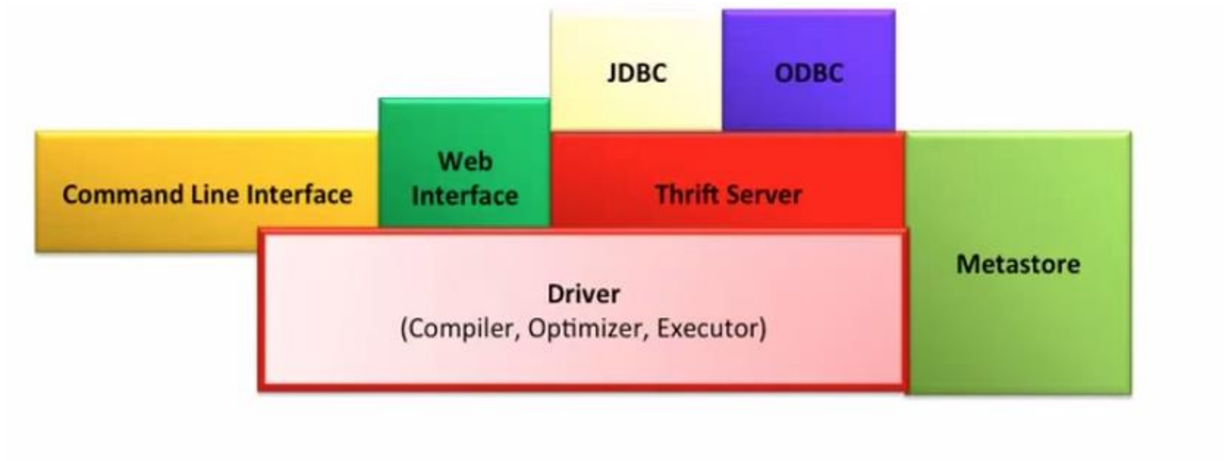
- Framework for cross language services
- Server written in Java
- Support for clients written in different languages
  - JDBC(java), ODBC(c++), php, perl, python scripts

# Metastore



- System catalog which contains metadata about the Hive tables
- Stored in RDBMS/local fs. HDFS too slow(not optimized for random access)
- Objects of Metastore
  - ➢ Database - Namespace of tables
  - ➢ Table - list of columns, types, owner, storage, SerDes
  - ➢ Partition – Partition specific column, Serdes and storage

# Hive Driver



- **Driver** - Maintains the lifecycle of HiveQL statement
- **Query Compiler** – Compiles HiveQL in a DAG of map reduce tasks
- **Executor** - Executes the tasks plan generated by the compiler in proper dependency order. Interacts with the underlying Hadoop instance

## Compiler

- Converts the HiveQL into a plan for execution
- Plans can
  - Metadata operations for DDL statements e.g. CREATE
  - HDFS operations e.g. LOAD
- Semantic Analyzer – checks schema information, type checking, implicit type conversion, column verification
- Optimizer – Finding the best logical plan e.g. Combines multiple joins in a way to reduce the number of map reduce jobs, Prune columns early to minimize data transfer
- Physical plan generator – creates the DAG of map-reduce jobs

## HiveQL

DDL :

- CREATE DATABASE
- CREATE TABLE
- ALTER TABLE
- SHOW TABLE
- DESCRIBE

DML:

- LOAD TABLE
- INSERT

QUERY:

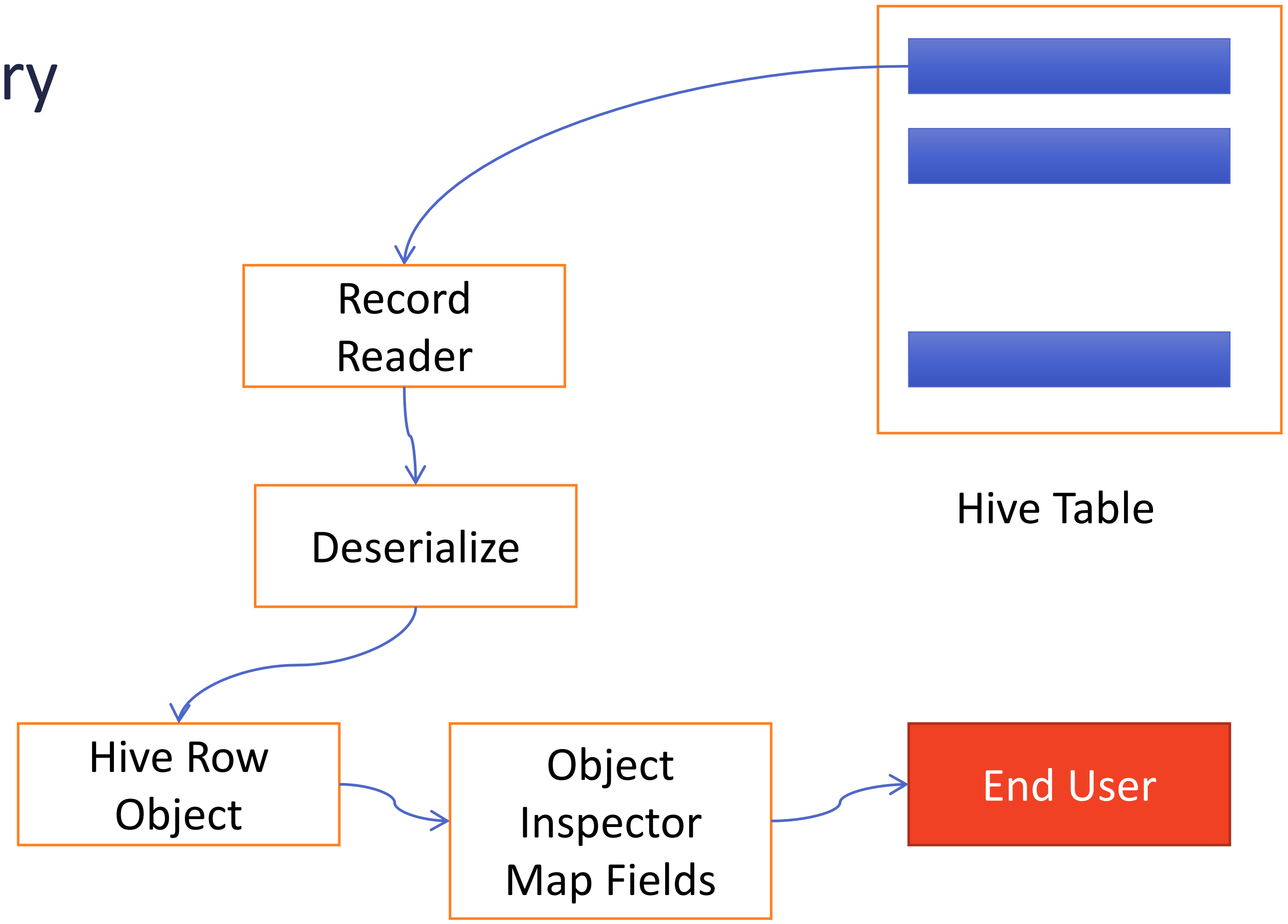
- SELECT
- GROUP BY
- JOIN
- MULTI TABLE INSERT

# Hive SerDe

- SELECT Query

➢ Hive built in Serde: Avro, ORC, Regex etc

➢ Can use Custom SerDe's (e.g. for unstructured data like audio/video data, semistructured XML data)



## Try on your Own

- **Use Hive and HiveQL with Hadoop in HDInsight**
  - https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-use-hive

- Thu Apr 13, TensorFlow by Narayan Hegde, Google
- 330PM, CDS102

## HBase

# GraphDB