COMP5349 – Cloud Computing

Week 10: Cloud-based NoSQL Databases

A/Prof Dr Uwe Röhm
School of Information Technologies

Outline

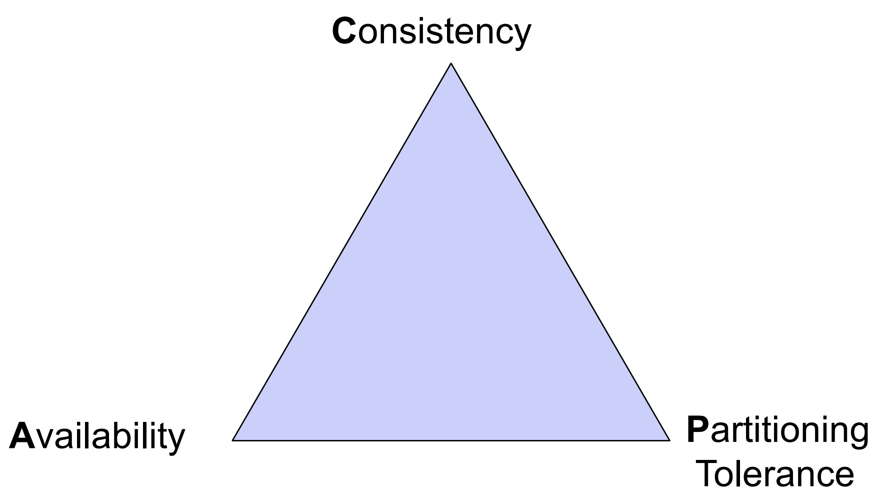
- Distributed Data Management
  - Main challenges
  - NoSQL Data Storage Overview
- Key-Value Stores
  - Amazon's Dynamo and Apache Cassandra
- NoSQL Column Stores
  - Google BigTable, HBASE

Goals for a Shared Data System?

- Scalability
  - Internet-scale systems must be able to scale fast to Petabyte level
- High Availability
  - A data system should always be up.
    E.g. Werner Vogels keynote: Amazon will always take your order
  - The challenge: the larger the system, the higher the prob of failures
- Partition Tolerance
  - If there is a network failure that splits the processing nodes into two groups that cannot talk to each other, then the goal would be to allow processing to continue in both subgroups.
- Strong Consistency
  - We would like to have 'all-or-nothing' semantics
    and ideally have all copies of the same data always consistent as if there is only a single copy

The CAP Theorem

[Brewer, PODC2000]



- Theorem:
  You can have **at most two** of these properties for any shared-data system.

NoSQL Data Storage

NoSQL or NoRel?

- Original idea: a lightweight DBMS that does not expose a full-fledged SQL interface
- Originally, MySQL was started with this in goal
  - But in the meanwhile, clearly moves towards a full-fledged DBMS including complex SQL, triggers, stored procedures etc.
  - Single outstanding feature though: supports different storage engines that can be used *per table*
- Today:
  Better name would be 'Non-Relational' as the most common interpretation of "NoSQL" is nowadays "non-relational"
  - or as some say: "Not-Only-SQL"

Criticisms about 'SQL' databases

- Heavy-weight and hard to understand
  - Yes, Oracle comes in *n* CDROMs
  - But then, sqlite is ~300KB and available in most Unix's by default
- "Slow"
  - What is the definition of 'slow'? (better: think scalability)
  - Yes, an SQL interface introduces some overhead,
    - but can help if used wisely – e.g. by pushing complex queries to DBMS
  - But most overhead is less about SQL, but how it is used + Transacts
- Expensive
  - Need qualified DBA personal; license costs of commercial DBMS…
- Schema must be known first
  - This is a valid point…
- We don't need transactions or strong consistency
  - depends – in general, it's simply the price people are Ok to pay atm.

The Different Flavors of NoSQL

- In the blue corner: Relational DBMS
  - Set of fixed-structured tuples; 1NF; joins; SQL; transactions
- In the red corner: 'NoSQL'
  - 'document-oriented' databases
    - Nested, hierarchical lists of key-value pairs
    - e.g. MongoDB or XML databases
  - No-SQL column stores
    - e.g. Amazon's SimpleDB, Google's BigTable, or HBase
  - Pure Key-Value stores
    - e.g. Amazon's Dynamo
  - Graph databases
    - E.g. Neo4J
  - …
- Note: So far, no standard data model or API for 'NoSQL'

NoSQL Classification Attempt

	SQL	Column Stores	Key-Value Stores
Data Model	Set of tuples	(key,column,value) triples	(key, value) pairs
Declarative Querying	SQL	Selections on keys, including scans	Select key
Updates	in-place update of single or tuple set	via creation of new column values	update(key: value)
Transactions	ACID	None	None; assumes single (k,v) access
Physical Design	Indexes, Partition, Mat. Views, …	Index on key	Index on key
Distribution	horizontal or vertical partitioning	horizontal and vertical partitioning	Partitioning by key
Replication	All kinds…	HDFS replication	eventual consistency
Examples	Postgres,MySQL, Oracle, DB2, SQL Server, Sybase	HBase, BigTable	Amazon Dynamo Cassandra

NoSQL Key-Value Stores Amazon Dynamo and Apache Cassandra

Example: Amazon's Dynamo

[SOSP2007]

- Highly scalable (key-value) store
  - Just two operations:
    - **Put**(*key*, *object*)
    - *object* **Get**(*key*)
- Core Assumptions
  - Simple read/write operations to data with unique IDs
  - No operation spans multiple entries
  - Data stored of small size
- Trade-off:
  - **Scalability** and **Availability** is everything (assume that errors happen)
    - Amazon really cares for 'write-is-always-possible'
  - Guaranteed SLAs
    - Goes back to Scalability: guaranteed response time for 99.9%requests
  - Consistency, Declarative Querying and general Transactions not as important in some scenarios

[following slides from SOSP2007 talk]

Summary of Techniques in Dynamo

Problem	Technique	Expected 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.

[SOSP2007, Table1]

Apache Cassandra

- Open-source distributed storage system
- Implements basically the Dynamo ideas, but with a more complex data model inspired by BigTable (cf. next section)
  - (*key*, *value*) pairs, where the values can have a nested sub-structure
  - *value* has several addressable columns, some of which are grouped to so-called 'column-families' (can be even nested)
- Developed at Facebook
  - phased out there around 2010
  - Now a top-level apache project
- Simple key-value store API
  - **Insert**(table, key, row)
  - **Get**(table, key, columnName)
  - **Delete**(table, key, columnName)

NoSQL Column Stores BigTable and HBASE

Google BigTable [OSDI2006]

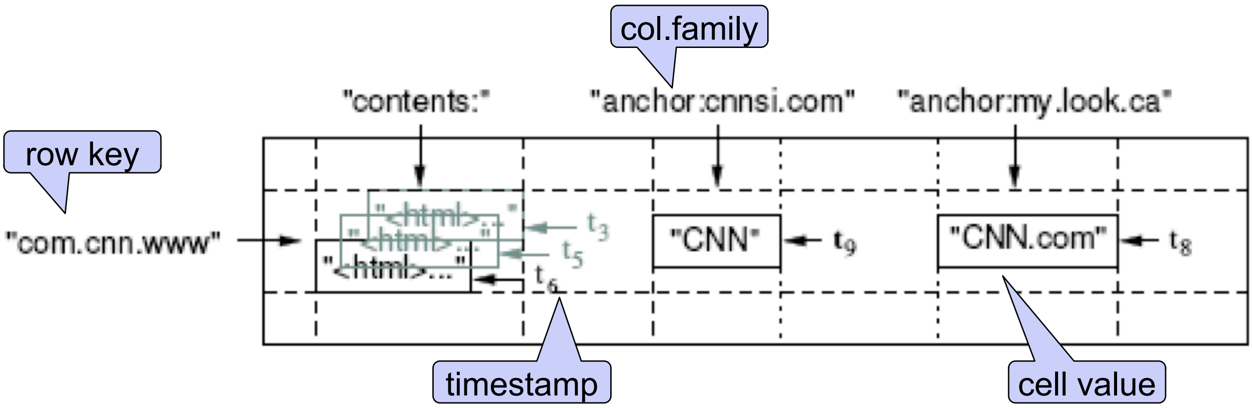
- Goals:
  - Reliable and highly scalable database for 'Big Data' (100s of TB – PB)
  - Designed to run on top of Google's Distributed FS
- Initially published in OSDI 2006.
- Benefits:
  - Distributed storage (parallelism)
  - Table-like data model
  - Highly scalable
  - High availability
  - High performance

HBase (http://hbase.apache.org/)

- A type of "NoSQL" database in the sense that it
  - It does not support SQL, nor transactions
  - In some sense, it is more a 'data store' than a 'database'
- A distributed, column-oriented database on top of HDFS
- Open-Source version of Google's 'BigTable'
  - Started end of 2006, became Hadoop sub-project in 2008, since 2010 Apache top-level project
- Difference to HDFS:
  - HBase allows fast lookup to single rows while HDFS does not interpret the content of its files.

Data Model

- Conceptually, HBASE' data model can be seen as big *tables*, which is made of *rows* and *columns*
  - Every row has a unique *row key*
  - Cells (intersection of rows and columns) are versioned (timestamp)
  - Only data type: (uninterpreted) *byte array*



[Source: Figure 1, BigTable paper, OSDI2006]

Data Model Implementation: Distributed Map of Triples

- (*row-key, column, timestamp)* triples as basis for lookup, inserts and deletes
- There can be an arbitrary number of “columns” per *row*
  - Columns are organised into *column-families*
  - Column-oriented physical storage – rows are sparse!

- (*row*:string, *column*:string, *time*:int64) → byte[]

Relational Data Model vs HBASE Model

web table

url	**language**	**content**
"www.cnn.com"	"EN"	"<html> … </html>"

link table

url	**referencingUrl**	**anchorText**
"www.cnn.com"	"connsi.com"	"CNN"
"www.cnn.com"	"my.look.ca"	"CNN.com"

column family

column family

column family with two column keys

Row key

	language	content	anchor	
			"connsi.com"	"my.look.ca"
"com.cnn.www"	"EN"	"<html> …"	"CNN"	"CNN.com"

Operations

HTable class methods:

- **Get** Returns attributes for a specific row
- **Put** Add new rows to a table or updates existing rows.
- **Scan** Allows iteration over multiple rows for specified attributes. Scan range is a row-key range
- **Delete** Removes a row from the table

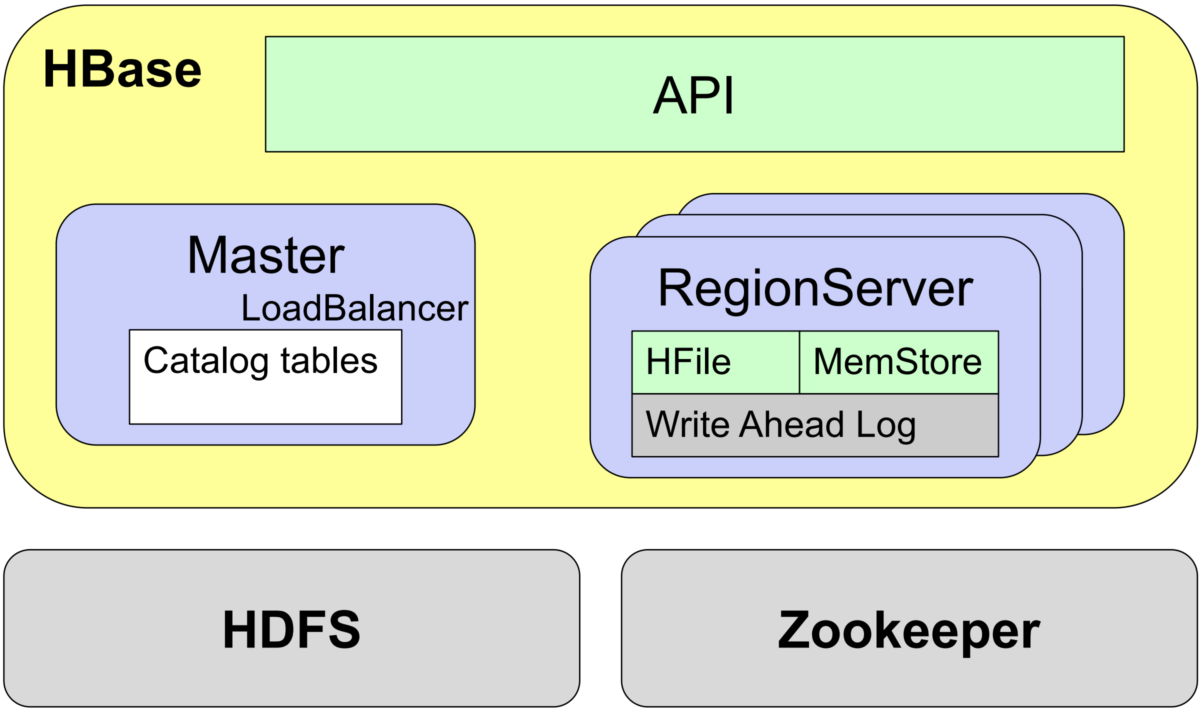
Physical Storage of HBase

- Tables are store in a per-column-family fashion
  - Empty cells are not stored at all
- Tables partitioned into ***regions***
  - Google's BigTable called these 'tablets'
  - Region defined by start & end row (sorted range of rows)
  - Regions are the unit for distribution around a cluster
- Each Region is made of multiple ***stores***
  - One *store* per each column-family of the tables in this region
  - *Store* consists of *StoreFiles* (*HFiles*, granularity for growth) and *MemStore* (in-memory cache for update logging)
    - Each StoreFile is written as a file into HDFS

HBase Implementation (1)

- Master, one or more RegionServers, & Clients
  - Single-master cluster
  - Build on-top of HDFS and Zookeeper
    - Fail-safety comes through HDFS' replication
- Cluster carries 0 to *n* labeled *tables*
- **Master** assigns table regions to RegionServers
  - Schema edits managed by Master
  - Reallocation of regions when crash
  - Lightly loaded
- **RegionServer** carries 0 to *m regions*
  - RegionServer keeps *commit log* of every update
  - Updates go first to regionserver'scommit log, then to Region
  - Regions periodically moved by Master's LoadBalancer

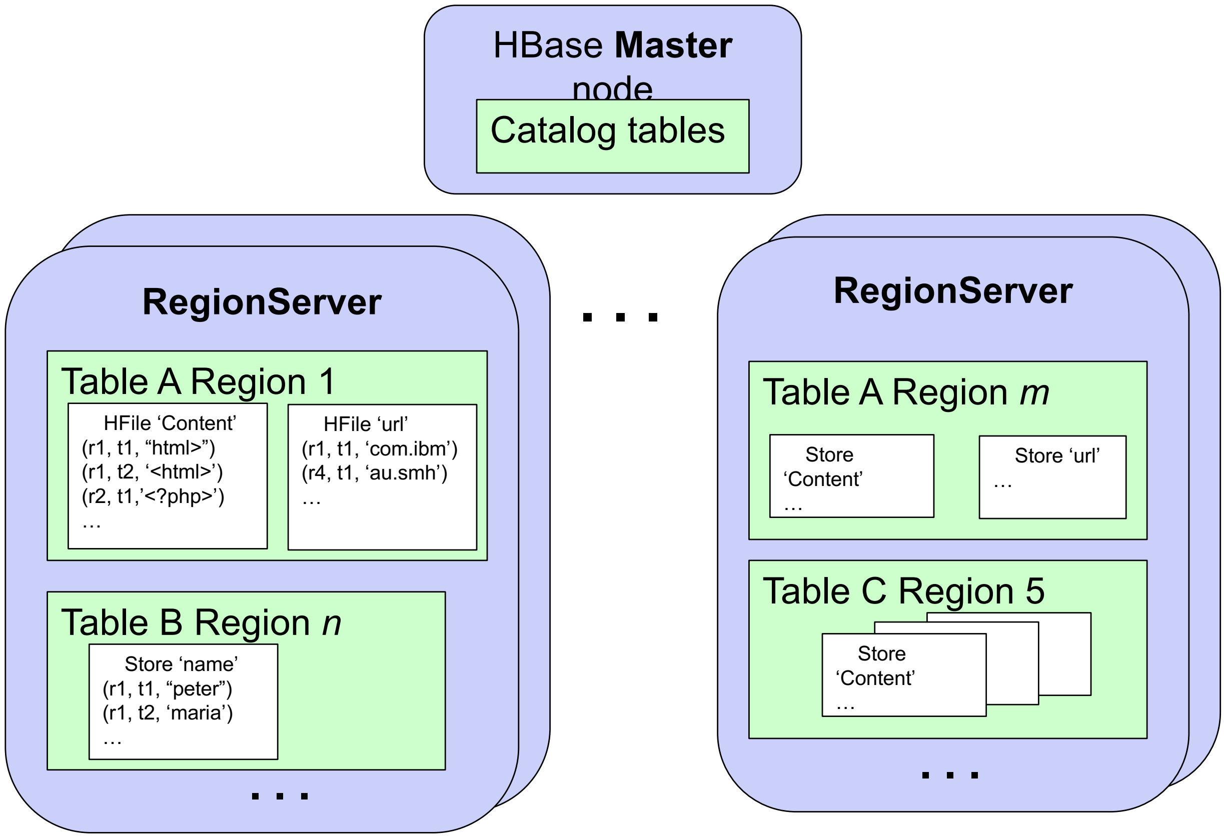
HBase Architecture



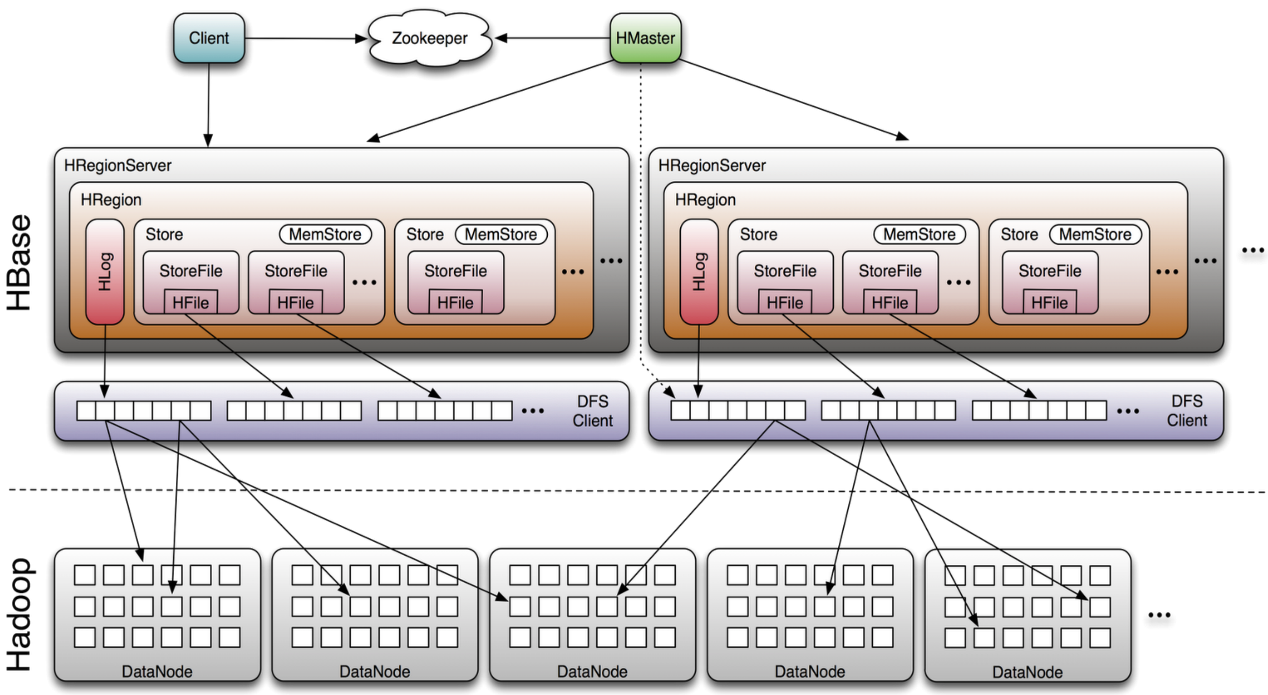
HBase Implementation (2)

- Each Region stores row ranges in a column-oriented way
  - Made of *stores*, one for each column family
    - All values for the same column family are stored together
    - Sparse rows: If a row has no value for a column family, then not present in the corresponding store (no need for NULL values)
      - Wide tables are fine
- Each store has an associated *MemCache* that takes on Region writes and which is flushed when full
  - Flush adds a *StoreFile*
    - Per Store, when > N StoreFiles, compacted in background
  - no in-place update or inserts to existing StoreFiles needed
    - Updates would not be supported by HDFS anyway
- When Regions get too big, they are split
  - Managed by RegionServer

System Overview



HBase System Architecture Details



http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

HBase Implementation (3)

- Request Flow
  - Client initially goes to Master for region hosting a row
  - Master supplies client a Region specification and host
  - Client caches this; goes direct to RegionServer thereafter
  - If fault (split/crash), returns to Master to freshen its cache
  - Region locations in *catalog* tables

- HBase is made of Hadoop Parts
  - Customized Hadoop RPC
  - MapFile for HStoreFiles
  - SequenceFile for commit logs, etc

Restrictions of HBase' Data Model

- Tables have just one primary index, the *row key*.
  - No other integrity constraints or secondary indexes
- No column typing
- No join operators
- Scans and queries can select a subset of available columns, even by using a wildcard.
- Three types of lookups:
  - Point-lookup using row key and optional timestamp.
  - Full table scan
  - Range scan from region start to end.

Restrictions of HBase' Data Model (2)

- Limited atomicity and transaction support.
  - HBase supports multiple batched mutations of single rows only.
  - Data is unstructured and untyped.

- Not accessed or manipulated via SQL.
  - Programmatic access via Java, REST, or Thrift APIs.
  - Scripting via JRuby.

Connecting to HBase

- Java client

```
HBaseConfiguration config = new HBaseConfiguration();
HTable table = new HTable(config, "myTable");
Cell cell = table.get("myRow",
                   "myColumnFamily:columnQualifier1");
```

- Non-Java clients
  - Thrift server hosting HBase client instance
    - Sample ruby, c++, c#!, & java (via thrift) clients
  - REST server hosts HBase client
    - JSON or XML
- Map/Reduce
  - TableInput/OutputFormat
    - HBase as MapReduce source or sink

Stream Messaging System Apache Kafka

Why Stream Processing?

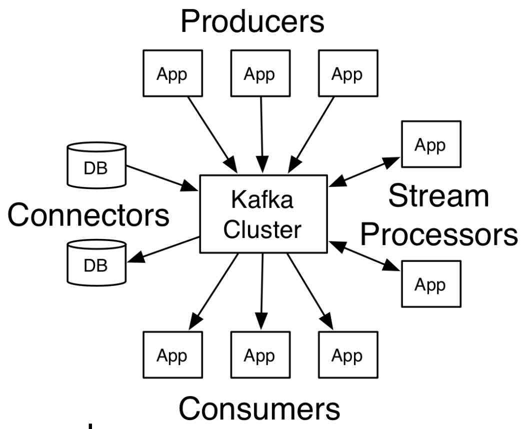
- Many applications with large streams of live data that needs to be processed immediately (‘real-time’)
  - Traffic management applications
  - Stock markets
  - Fraud detection in payment systems
  - Wireless sensor networks
  - Environmental monitoring systems

- Why not a DBMS?
  - Not agile enough; needs persistent storage and indexing before processing
  - “data at rest”; favours infrequent updates
- Stream Processing Systems
  - “Data in motion”: Processing data as it flows without storing persistently
  - But may buffer temporarily to provide windows

Overview of Kafka

https://kafka.apache.org/documentation.html

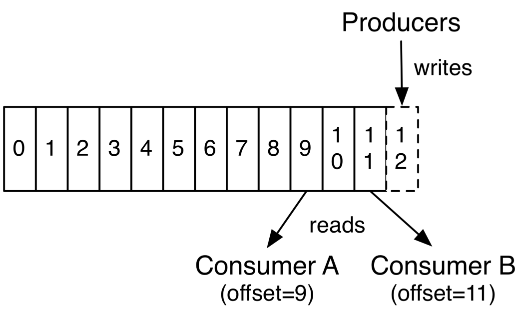
- Kafka is a distributed publish-subscribe messaging system
  - maintains feeds of messages from one or more *producers*
  - feeds can be subscribed to by *consumers*
  - Apache Flink and Spark often used as stream processors



- Three key capabilities
  1. publish and subscribe to streams of records
  2. can store streams of records in a fault-tolerant way
  3. lets Apps process streams of records as they occur

Data Model for Stream Processing

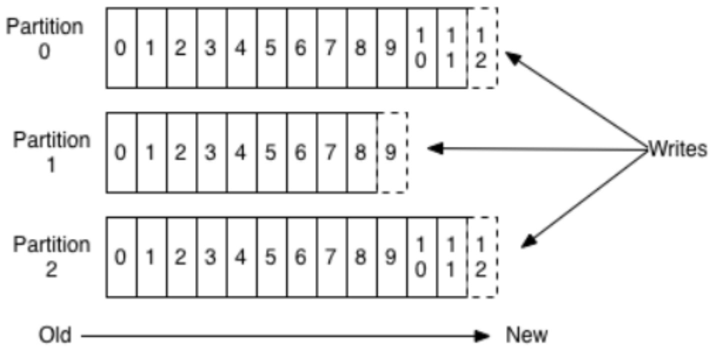
- Producers publish messages to topics (or feed) of their choice.
- Data stream is unbound and broken into a sequence of individual data records (aka messages).
- A record is the atomic data item in a data stream
  - similar to a structured commit log with append-only semantics



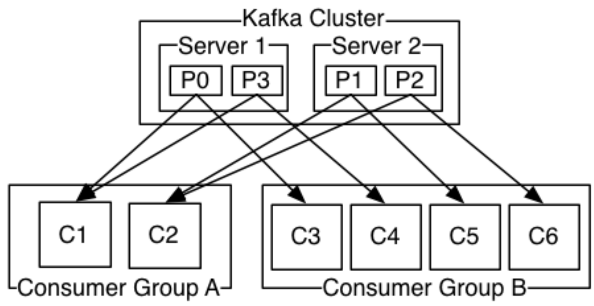
- Programming Model: Dataflow Programming
  - Producer – Transformation – Sink

Kafka Topics

- Kafka maintains a partitioned log for each topic



- Log is replicated across multiple servers
- Topics in Kafka are multi-subscriber
  - each record published to a topic is delivered to one consumer instance within each subscribing *consumer group*.
  - Consumer instances can be in separate processes or on separate machines.



Kafka as Storage System

- Kafka allows to decouple the consuming of data from the data producers
  - to do so, it implements a very efficient storage system
- Data written to Kafka is written to disk and replicated for fault-tolerance.
  - Kafka allows producers to wait on acknowledgement so that a write isn't considered complete until it is fully replicated.
  - Kafka prides itself that its disk structures scale well.
  - One can think of Kafka as a kind of special purpose distributed filesystem dedicated to high-performance, low-latency commit log storage, replication, and propagation.

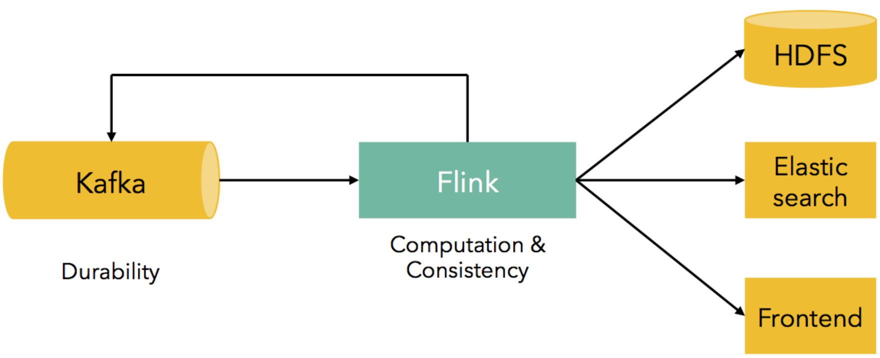
Guarantees from Kafka

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.
- A consumer instance sees records in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

- Message Delivery semantics
  - *At most once*: every message is processed at most once, but no guarantee
  - *At least once*: no message will get ignored
  - *Exactly once*: system guarantees that every meesage processed exactly once
  - Kafka: **at least once** semantics
    - as published messages are first 'committed' to aa topic log, it is guaranteed to not get lost
    - but depending on how consumer handles crashes, a message might be read more than once due to retries after a consumer failure

Data Stream Processing

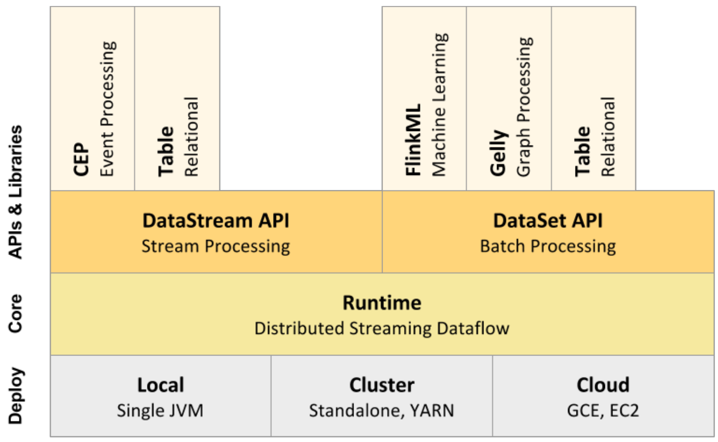
- Kafka itself is not a data stream processor
  - it is the messaging and storage system for data streams
- Common data stream processors on top of Kafka are Apache Spark and Apache Flink
  - Example with Flink:



[https://data-artisans.com/blog/kafka-flink-a-practical-how-to]

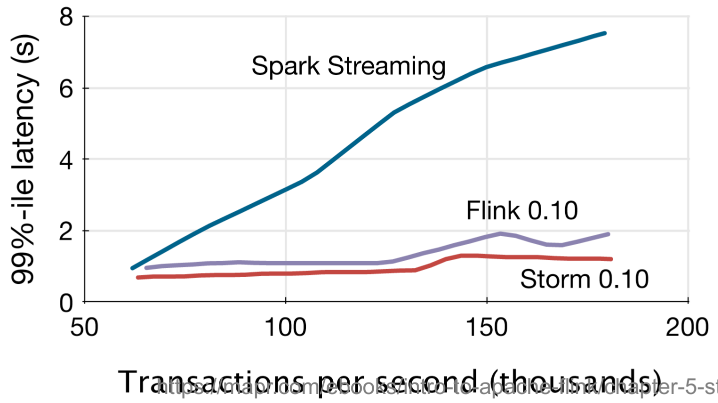
Data Stream Processing with Flink

- Note that Flink has two different APIs
  - DataStream API vs. DataSet API
- In our labs and the assignment, we are using the DataSet API
- But Flink lends itself very well to continuous stream processing because it uses pipelining throughout the whole system
  - tuples are immediately forwarded and consumed by next operator if available
  - major difference to Apache Spark, where processing stages are clearly separated



Differences between Spark and Flink

	Apache Spark	Apache Flink
Principle	set-oriented data transformations in stages	transformations by iterating over collections with pipelining
Data Abstraction	RDD	DataSet
Processing Stages	separate stages	overlapping stages
Optimiser	with SparkSQL	integrated into API
Batch Processing	RDD	DataSet
Stream Processing	micro-batching	pipelining; DataStream API



https://mapr.com/ebooks/intro-to-apache-flink/chapter-5-stateful-computation.html

Summary

- **Background**
  - **CAP Theorem**
    - In large distributed systems, at most 2/3 DAP properties achievable
  - **Data Replication** and **Data Partitioning**
- **NoSQL Storage Systems**
  - Non-SQL data model + CAP principles
  - Key/value Stores: Amazon's **Dynamo** and Facebook's Cassandra
  - Column Stores: Google's BigTable and Hadoop's **HBASE**
- **HBase**
  - Open-source implementation of BigTable
  - Distributed multi-dimensional map
  - Implemented with Master, that manages multiple RegionServers; each RegionServer holds multiple table regions that are stored in a column-oriented way on top of HDFS
- **Apache Kafka**: stream-oriented messaging systems

References

- *Dynamo: Amazon's Highly Available Key-value Store.* Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. SOSP 2007.
- *Cassandra - A Decentralized Structured Storage System.* Avinash Lakshman and Prashant Malik. LADIS 2009.
- *BigTable: A Distributed Storage System for Structured Data.* Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah W. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Rober E. Gruber, OSDI 2006.

- http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext

Next Week

- **Data Consistency and Cloud Computing**
  - Data Replication & Partitioning
  - Data Consistency Notions
  - Paxos Protocol

- **Readings:**
  Werner Vogels: Eventually consistent. (CACM, 2009)
  Leslie Lamport: Paxos Made Simple. (2001)