HBase (Wide-Column Store)

Data Model: Wide-Column Store

- Modeled after Google BigTable
  https://static.googleusercontent.com/media/research.google.com/en//archiv
  osdi06.pdf
- Part of Apache Hadoop
- Runs on top of HDFS (Hadoop Distributed File System)
- Supports Hadoop Map/Reduce
- CAP-Theorem: CP-System
- Interface: Java, REST, Thrift, HBase Shell

HBase (Wide-Column Store)

Data Model: Table with row-id plus column families that contain arbitrary columns.

Value Data Type: byte[]

```
> create 'websites', {NAME=>'info'}, {NAME=>'links'}
> put 'websites', 'www.othr.de', 'info:title', 'OTH Regensburg'
> put 'websites', 'www.othr.de', 'info:language', 'German'
> put 'websites', 'www.othr.de', 'links:www.stwno.de', ''
> get 'websites', 'www.othr.de'
```

Row-ID	info		links
www.othr.de	**title**	**language**	**www.stwno.de**
	OTH Regensburg	German	

Each row of a table in a wide-column store can contain arbitrary columns. To separate different concepts in the same table, column families are defined at table-creation time. Here, we separate the infos of a web site (other web sites can have similar or other columns) and a set of outgoing links. HBase only supports methods for accessing a row by its row-id or scanning over a set of rows by specifying a row-id range. Rows are distributed on cluster nodes using range partitioning on the row-ids. HBase stores its data in the Hadoop distributed file system. MapReduce is often used to execute complex computations on big data.

Versioning in HBase

HBase Data Model:

$$Table \to (RowID \to (ColFamily \to (Col \to (Timestamp \to Value))))$$

- Each column can contain multiple version of its values (history)
- Each value is annotated with the timestamp of its last write
- Number of versions can be specified for each column family (default: 1)

```
> CREATE 'websites', {NAME=>'info', VERSIONS=>9}, 'links'
```

```
Configuration config = HBaseConfiguration.create();
Connection conn = ConnectionFactory.createConnection("config");
Table table = conn.getTable(TableName.valueOf("websites");
Get g = new Get("www.othr.de");
Result r = table.get(g);
//  colFam      column      time value
Map<byte[],Map<byte[],Map<Long,byte[]>>> = r.getMap();
```

The example shows Java code to access all columns in all column families of the row with row-id `"www.othr.de"`. For each column, we have a map of multiple (here: up to 9) versions. There are other methods to only access specific column families, only specific columns, only the current version, only the last n versions, etc.

HBase - Example Application

Row-ID	**Log**				
user5/1619512603	**action**	**type**	**query**	**size**	**ip**
	search	image	cats	large	82.202.31.71
user8/1619522109	**action**	**post-id**	**text**	**ip**	**mobile**
	comment	923921	Okay	82.202.31.71	Android 11

This design of the row-id allows for queries like:

```
// the 10 most recent activies by user 5
scan = new Scan();
scan.withStartRow(toBytes("user6/"));
scan.withStopRow(Bytes.toBytes("user5/"));
scan.setReversed(true);
scan.setMaxResultSize(10);
```

As HBase is optimized for GET and SCAN requests based on the row-id or ranges of row-ids, it is important to choose the row-id so that it fits to typical queries. In this example, we log each user's action on a web site in an HBase table. The row-id consists of a user id and a timestamp. We do a range search on the row-id to find every row with the row-id prefix `user5/`.

HBase - Example Application

Row-ID	**Log**			
search/1619512603/user5	**type**	**query**	**size**	**ip**
	image	cats	large	82.202.31.71
comment/1619522109/user8	**post-id**	**text**	**ip**	**mobile**
	923921	Okay	82.202.31.71	Android 11

This design of the row-id allows for queries like:

```
// who logged in at 27th April 2021?
scan = Scan(Bytes.toBytes("login/1619481600"),
            Bytes.toBytes("login/1619567999"));
```

Here, we store the very same data as on the slide before. But this time, the row-id consists of the name of an activity, a timestamp, and the user. We can now use a scan to find all activities of a specific type (e.g. login) within a given time range.

Apache Cassandra

Data Model: Wide-Column store

CQL3 (Cassandra Query Language) for quering the database.

SQL style language parts:

- Data Definition Language (DDL)
  Altering the database schema (meta-data)
- Data Modification Language (DML)
  Modifying the data

SQL-style tables and queries are mapped to the Cassandra Storage Engine.

Previous API: Thrift (deprecated)

Apache Cassandra

Data Definition Language:

A Keyspace contains a set of tables. Create one replica in Datacenter 1, three replicas in Datacenter 2

```
CREATE KEYSPACE webshop WITH replication =
         {'class': 'NetworkTopologyStrategy', 'DC1': 1, 'DC2': 3}
```

```
CREATE TABLE products (client int, pid int, name text,
        price DECIMAL, stock int, PRIMARY KEY (client, pid));
```

First component of the primary key: **Partition Key**
Second to nth component: Cluster keys (specify order inside partition)
Important for data distribution, searching and sorting

Choice of partition key influences efficiency of data distribution!

Apache Cassandra

Data Definition Language:

```
CREATE TABLE customers (client int, cid int, name text,
    categories set<text>, notes map<int,text>,
    clientname text STATIC,
    PRIMARY KEY (client, cid));
```

Composite Data Types:

- set<type>
- map<type,type>

Static columns: Values are shared with all rows containing the same partition key

Apache Cassandra

Difference to relational model:

```
CREATE TABLE users (userid int, username text STATIC,
                    category text, property text, value text,
                    PRIMARY KEY (userid,category,property));
```

userid	username	category	property	value
27	anna	gui	color	red
27	anna	auth	lastlogin	2022-05-23
28	paul	gui	theme	dark

Projections of a wide-column similar to:

userid	username	gui:color	auth:lastlogin
27	anna	red	2022-05-23

userid	username	gui:theme
28	paul	dark

Apache Cassandra

Data Modification Language:

SELECT queries the database, predicate should match table clustering

```
SELECT username,property,value FROM users
                     WHERE userid = 27 AND category > 'f';
 username | property | value
----------+----------+--------
     anna |    color |   red
```

Difference to HBase: No efficient range queries for row key because of hash partitioning

Clustering keys: Allows efficient range queries if all preceding clustering keys are equality restricted

Apache Cassandra

Data Modification Language:

INSERT inserts or updates data:

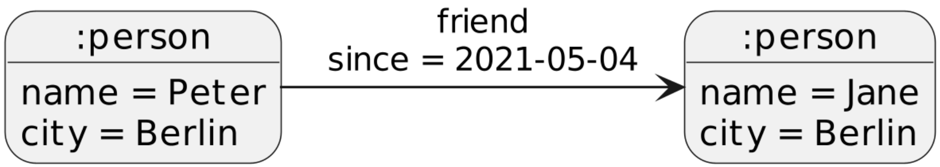
```
INSERT INTO users (userid,category,property,value)
    VALUES(27, 'gui', 'color', 'green');
INSERT INTO users (userid,category,property,value)
    VALUES(27, 'gui', 'theme', 'light');
SELECT username,property,value FROM users
                     WHERE userid = 27 AND category > 'f';
 username | property | value
----------+----------+-------
     anna |    color | green
     anna |    theme | light
```

Neo4J (Graph Database)

Data Model: Property Graph $G = (V, E)$

- *Vertices*: have a label (node type) and a set of properties (key-value pairs)
  `(:person { "name": "Peter", "city": "Berlin" })`

- *Edges*: directed connection between two nodes, have a label and properties
  `() -[:friend {"since":"2021-05-04"}]-> ()`



Graphs consist of nodes edges. In typical graph databases, both nodes and edges can have a label, and a list of arbitrary properties. In Neo4J, there are only directed edges. So, when we want to model a friendship between two person nodes - friendship is a symmetric relationship -, we simply choose an arbitrary direction, either an outgoing or incoming edge. Later, when querying the data, we simply traverse edges in both directions to find a person's friends.

Neo4J (Graph Database)

Some typical graph operations:

- Find connections (of specific types) from a node to other nodes
- Find connections of degree **n**
- Find shortest path between two nodes
- Find connected components

Neo4J - Cypher (Query Language)

`MATCH` clause: Pattern matching within the graph

`(variable : node_label) -[variable : edge_label]-> ()`

`WHERE`, `ORDER BY`, `LIMIT`, `RETURN`, `UPDATE`, ... clauses: as in SQL or XQuery

```
MATCH (p : person)-[:friend]-(x : person)
WHERE p.name = 'Peter'
ORDER BY x.name
RETURN x.name, x.city
```

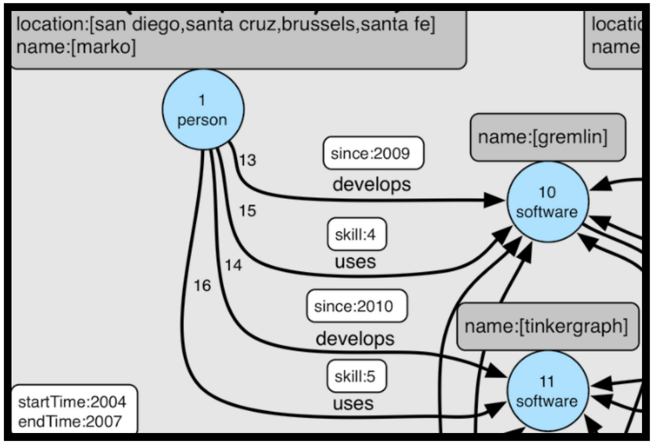
Within a MATCH clause `()` represent a node, and `--` / `-[]-` is an edge. Edges can have a direction `-->` / `<--`. If no direction is specified `--`, both incoming and outgoing edges are matched. It is possible to introduce variables (here: p and x) to refer to a node or edge, e.g. within a `WHERE` predicate or the `RETURN` clause. After a colon, the node or edge label can be specified, e.g. `:person`. The query on this slide finds the names and cities of Peter's friends.

Gremlin (Graph Query Language)

```
gremlin> graph = TinkerFactory.createTheCrew()
gremlin> g = traversal().withEmbedded(graph)

// find Marko's person node
gremlin> g.V().has('person', 'name', 'marko')
==>v[1]

// which software did Marko develop?
gremlin> g.V().has('person', 'name', 'marko')
         .out('develops').hasLabel('software')
         .values('name')
==>gremlin
==>tinkergraph
```



Full graph at https://bit.ly/thecrewgraph

Gremlin is a universal graph query language. Here, we use it to query an example graph within the embedded im-memory graph database Tinkergraph. The first two lines in the code creates a the example graph and initializes a traversable object g. With Gremlin, we traverse the graph by navigating over nodes and edges. In the longer example query on this slide, we first find all nodes `g.V()`, then we select only the person node with name Marko. After that we navigate to its neighbor nodes via outgoin `develops` edges, check wether their label is software, and finally return the value of the `name` property of that nodes.