GRAPH, COLUMN, AND IN MEMORY DATA BASES

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Graph Databases (1)

- DB designs disagree on various aspects of design, **but:**
  - Databases store information about “things” and their relationships!
  - Relationships have as natural counterpart: **Graphs**

- Relational Databases can easily model graphs, **but**
  - *RDBMS generally hits performance issues with large graphs!*
  - *SQL might be a little limiting (no full recursion!)*

- NoSQL are even worse at modeling graphs
  - Graph structure can be stored in a single document or object**, but**
  - *relationships between objects are not inherently supported (no joins!)*

- The need for a different DBMS
  - The Graph database system

RDBMS Patterns for Graphs

- The relational model can easily represent the data that is contained in a graph model, but
  - SQL lacks the syntax to perform graph traversal
  - Each level of traversal adds significantly to query response time

```
SELECT p2.personname, m1.movieName
  FROM people p1
       JOIN actors a1 ON (p1.personid = a1.personid)
       JOIN movies m1 ON (a1.movieid = m1.movieid)
       JOIN actors a2 ON (a2.movieid = m1.movieid)
       JOIN people p2 ON (p2.personid = a2.personid)
 WHERE p1.personname = 'Keanu Reeves';
```

Figure 5-3. *SQL to perform first-level graph traversal*

Graph Databases (2)

- A Graph is given by
  - A set of Nodes (or vertices) **N** and
  - A set of edges (or arcs) **E** modeling associations between two nodes.
  - **N** and **E** can have associated properties (labelled graphs)

- There is mathematical notation for performing operations
  - Add or remove nodes and edges or find adjacent nodes, etc.

- "Queries" end up in performing a *graph traversal*
  - *i.e.*, walking through the graph to explore the graph

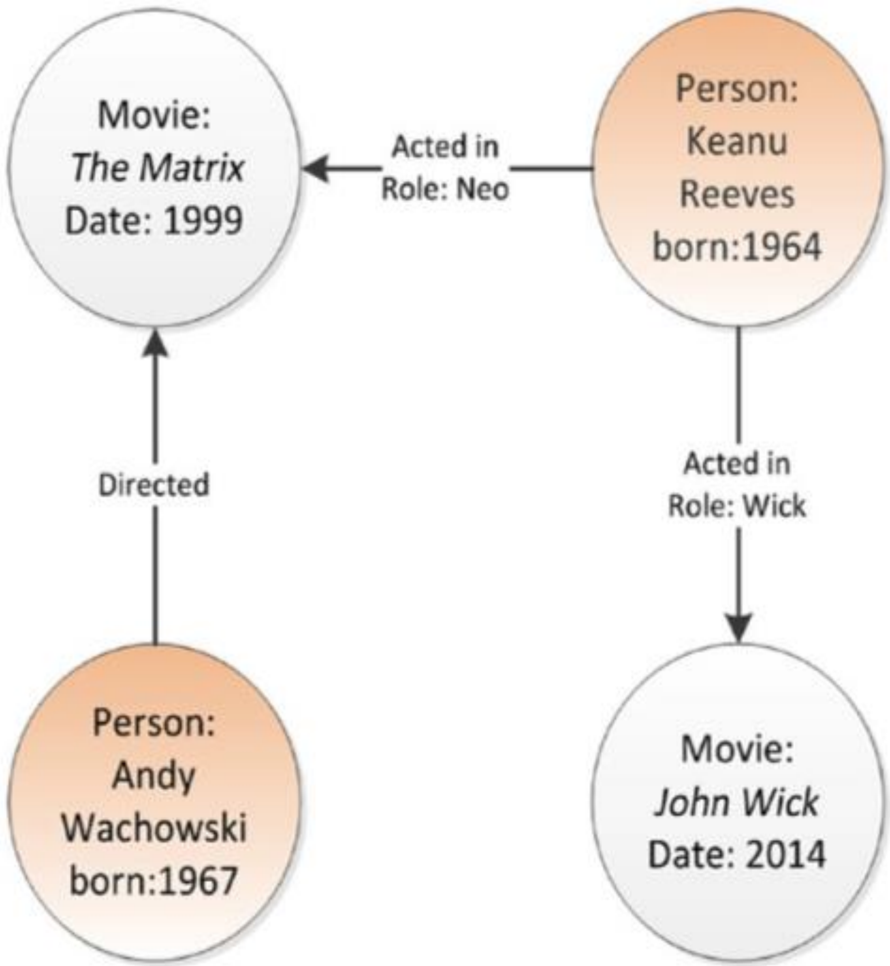


Figure 5-1. *Simple graph with four vertices (nodes) and three edges (relationship*

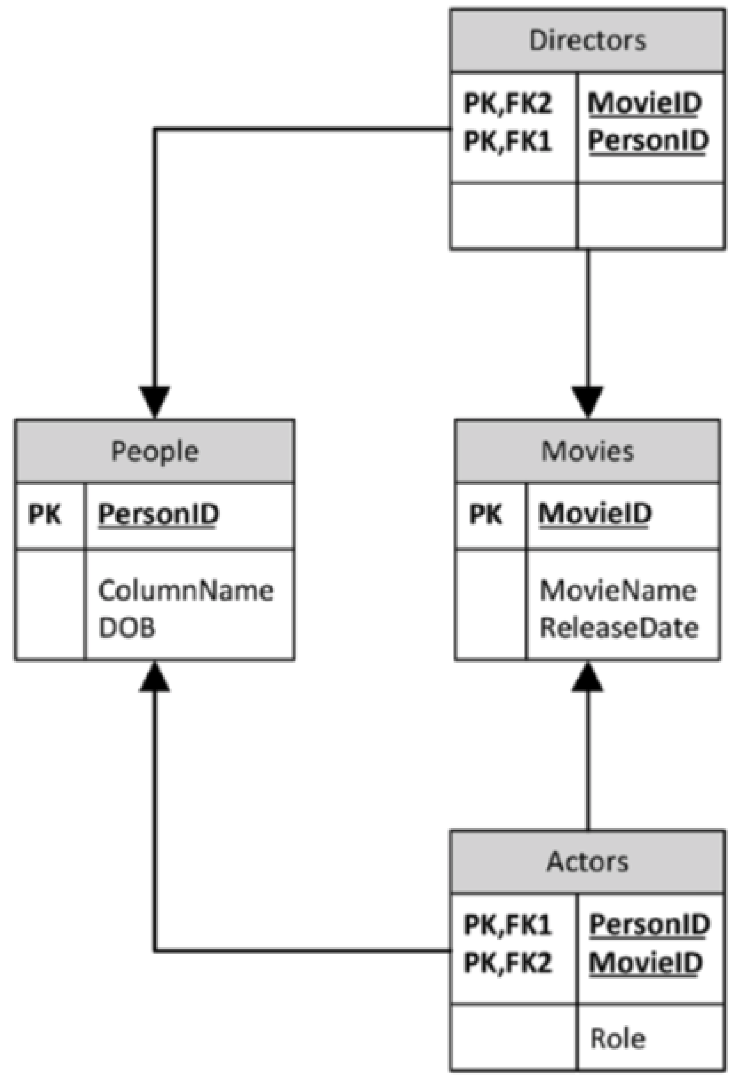


Figure 5-2. *Relational schema to represent our sample graph data*

Property Graphs and Neo4j

- *Property Graph* model
  - Associates both nodes and relationships with attributes
- *Neo4j*
  - Probably the most widely adopted graph database
  - *Cypher:* a declarative graph query language
    - Better than SPARQL to specify graph traversals

```
neo4j-sh (?)$ MATCH (kenau:Person {name:"Keanu Reeves"})
>             RETURN kenau;
```

  - *Gremlin:* procedural graph database query language

Graph Database Efficienct Structures

- *Index-free adjacency*
  - Enabling efficient real-time graph processing
  - Move through the graph structure without index lookups
  - Each node knows the physical location of all adjacent nodes
    - No need to use indexes to efficiently navigate the graph

- Not easy to distribute
  - The overhead of routing the traversal across multiple machines eliminates the advantages of the graph database model
  - Inter-server communication is far more time consuming than local access
  - Neo4j do not currently support a distributed deployment

Graph compute engines

- Graph processing and other data models work across massive distributed datasets
  - **Apache Giraph**
    - Designed to run over Hadoop data using MapReduce
  - **GraphX**
    - Part of the *Berkeley Data Analytic Stack* (*BDAS*)
    - Uses Spark as the foundation for graph processing
  - **Titan**
    - A scalable graph database optimized for storing and querying graphs distributed across a multi-machine cluster
    - Can be layered on top of Big Data storage engines, including Hbase and Cassandra

RDF

- The *Resource Description Framework* (*RDF*)
  - A web standard developed in the late 1990s
  - Information expressed in *triples*
    - entity: attribute :value
  - Intended for creating a formal database of resources
- RDF specification included XML syntax
- Native RDF databases are called *triplestores*
  - AllegroGraph, Ontotext GraphDB, StarDog, Virtuoso, and Oracle Spatial
- The *SPARQL* query language (a SQL-like language)

```
SELECT ?object
  FROM <http://dbpedia.org>
 WHERE { <http://dbpedia.org/resource/Edgar_F._Codd>
          <http://purl.org/dc/terms/subject> ?object }
```

Column Databases

Those of us raised in Western cultures have been conditioned to think of data as arranged in rows
[Guy Harrison]

- The first databases addressed OLTP processing
  - CRUD operations were the most time-critical ones
  - Row-based storage (record-oriented) provided good performance
- Analytic and decision support applications
  - Data warehousing and analytic workloads
    - Long batch processing and aggregations
  - *OLAP* (*Online Analytic Processing*)
    - Coined by Edgar Codd to differentiate from OLTP systems

OLTP vs OLAP

- OLTP system
  - ACID transactions and CRUD operations
  - Service-level response times
  - Row-oriented physical organization ideal (write-intensive)

- OLAP system
  - IO intensive aggregate queries
  - *New non-normal-form schemas: Star & Snowflake*
  - Column-oriented physical organization ideal (read-intensive)

- Can we support both?
  - Analytic and decision support applications with interactive response times
  - C-Store (http://db.lcs.mit.edu/projects/cstore/vldb.pdf)

Star schema

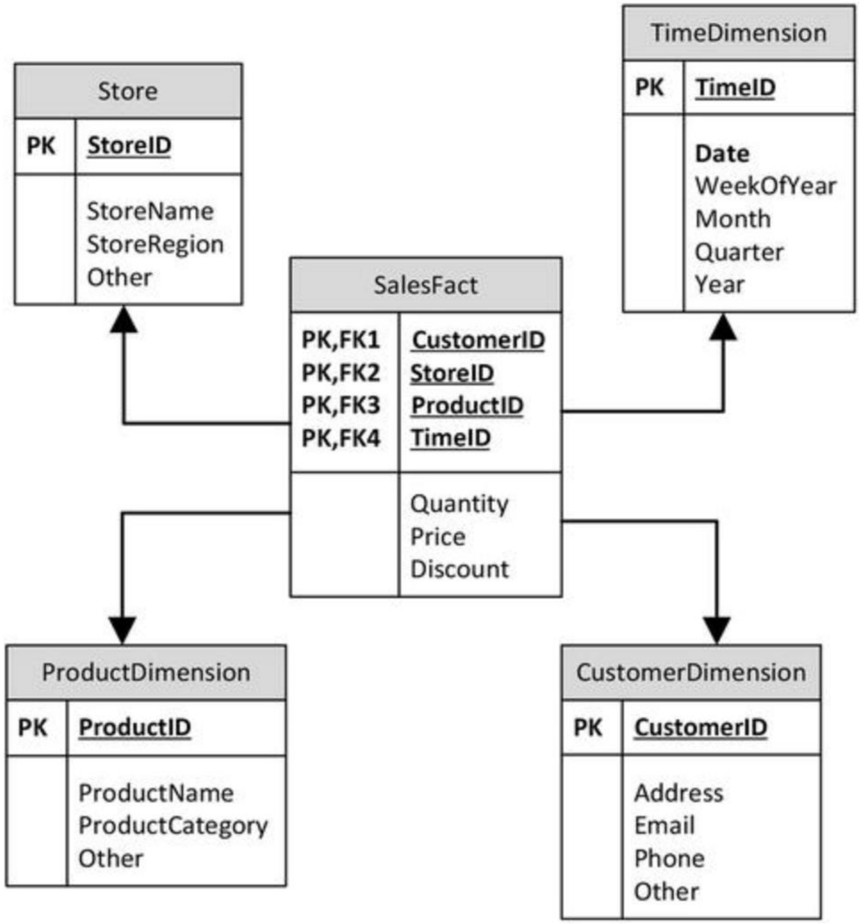


Figure 6-1. Star schema

- A schema for data warehouses:
  - Aggregate queries should execute quickly
- Central large *fact* tables are associated with
- Numerous smaller *dimension* tables
- Not a normalized relational model of data…

Star schema and scalability

- Almost all data warehouses adopted some variation on the star schema paradigm
- Almost all DBMS adopted indexing and SQL optimizations to accelerate queries against star schemas
- However, *processing in data warehouses remained severely CPU and IO intensive*

The columnar Alternative

- Data for columns is grouped together on disk
- Clear advantage in aggregate queries, but:
- Retrieving a single row involves assembling the row from each of the column stores for that table

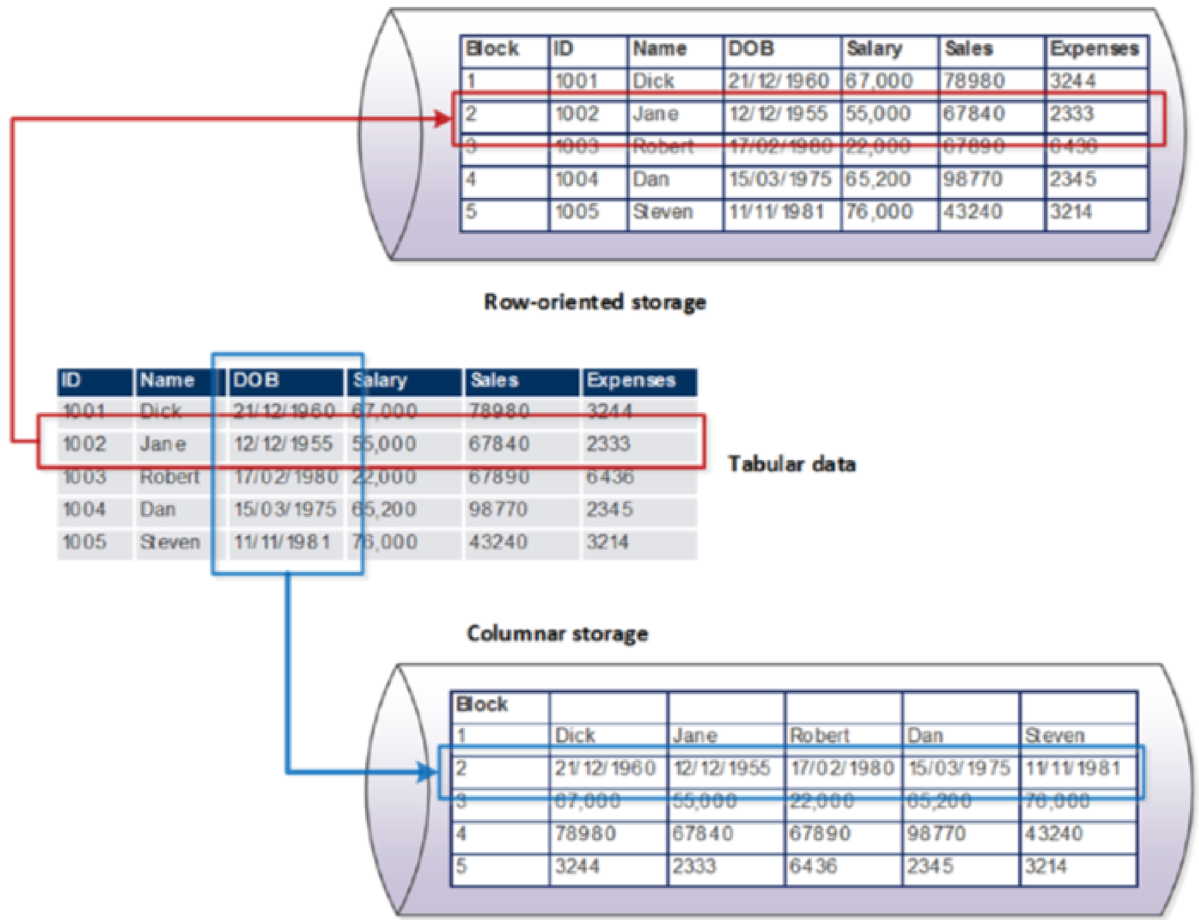


Figure 6-2. *Comparison of columnar and row-oriented storage*

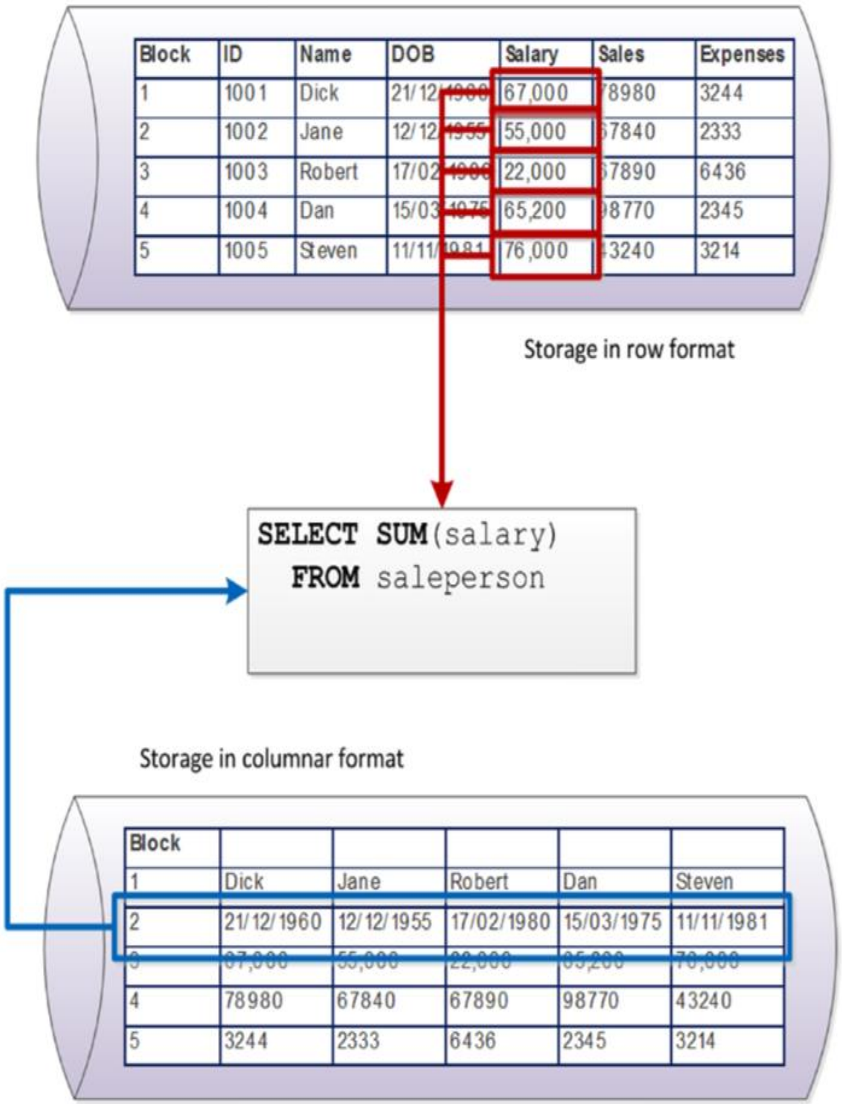


Figure 6-3. *Aggregate operations in columnar stores require fewer IOs*

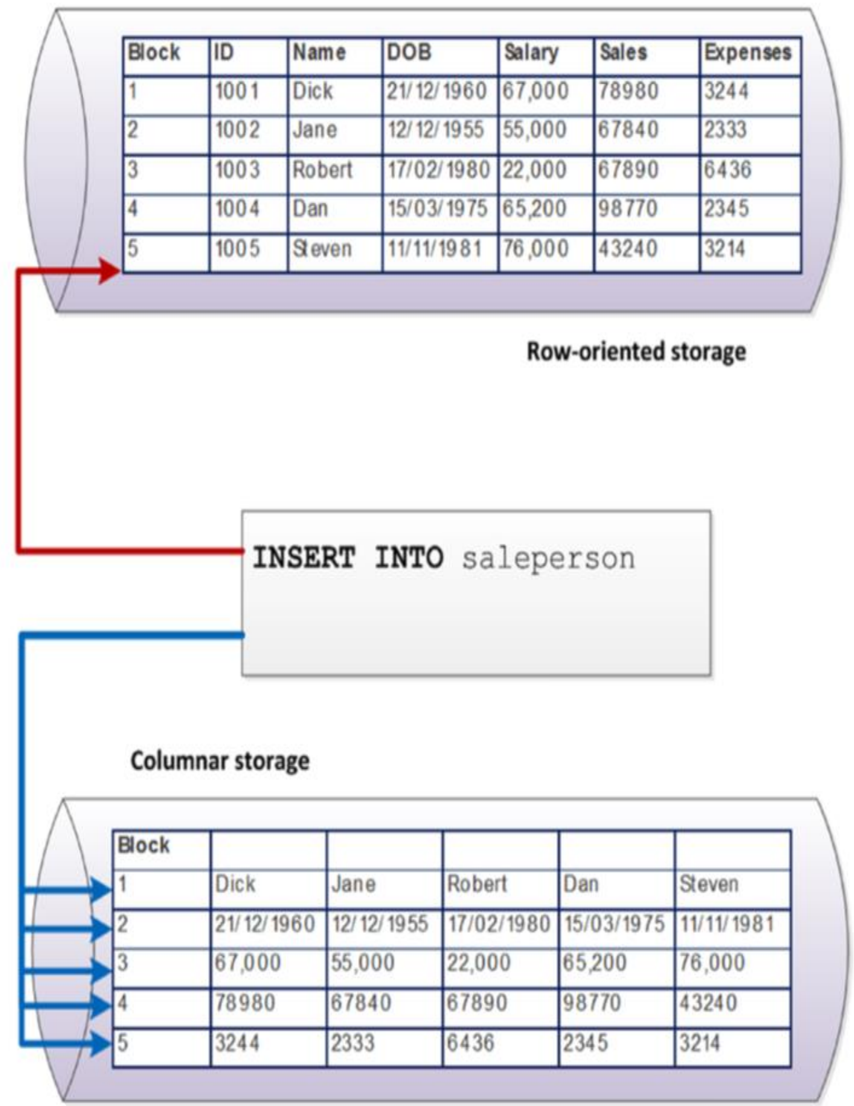


Figure 6-4. *Insert overhead for a column store*

Historical perspective (1)

- In the mid-1990s, Sybase acquired Expressway
  - The first significant commercial column-oriented database
  - Expressway became *Sybase IQ* (“Intelligent Query”)
    - *Sybase IQ failed to dominate the data warehousing market*
- In 2005, Mike Stonebraker and colleagues present *C-store*
  - A novel *column-oriented* DBMS
  - Faster than row-based DBMS for data warehousing

Historical perspective (2)

- Stonebraker founded *Vertica*
  - Acquired by HP in 2011

- Column-based systems entered the market
  - InfoBright, VectorWise, and MonetDB.
  - Oracle Exadata, Microsoft SQL Server, and SAP HANA

- Column-based systems
  - Significantly different from traditional RDBMS
  - Still based on SQL -> “NewSQL”

C-Store storage (1)

- Redundant storage of elements
  - Table split in overlapping *projections* in different orders

- Projections
  - Store combinations of columns together on disk
  - Combinations of columns that are frequently accessed together
  - Possibly sorted w.r.t. a different attribute
  - Redundant, but very efficient

- A set of projections covers the entire table
  - Additional projections are created to support specific queries

Region	Customer	Product	Sales
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Logical Table
Table appears to user in relational normal form

Physical Storage

Region	A	B	D	C	A
Customer	G	C	F	C	R
Product	C	C	D	A	B
Sales	789	743	675	23	654

1 **Super-Projection**
Default projection unsorted columnar storage of all columns. Used to resolve queries that have no specific supporting projection

Region	A	A	B	C	D
Product	B	C	C	A	D
Sales	654	789	743	23	675

Sort Columns

2 **Projection 1**
This might support query:

```
SELECT Region, Product, SUM(Sales)
 GROUP BY Region, Product
```

Customer	C	C	F	G	R
Sales	743	23	675	789	654

Sort Column

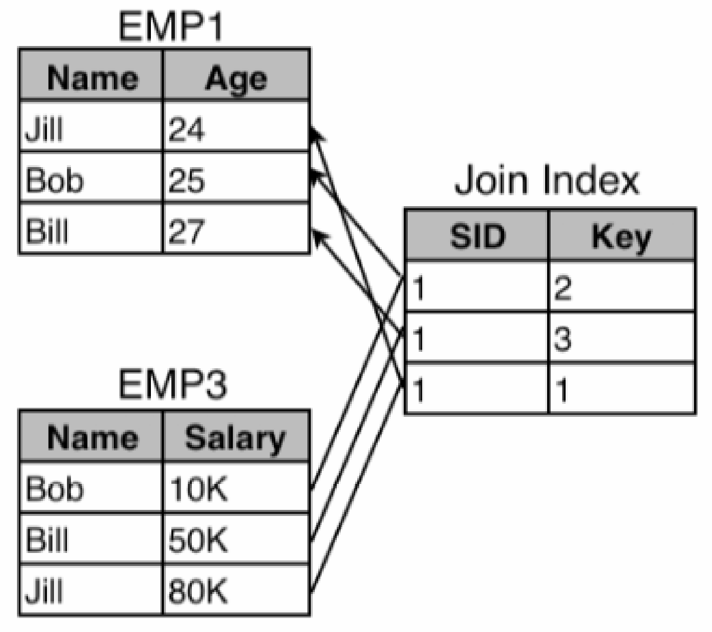
3 **Projection 2**
This might support query:

```
SELECT Customer, SUM(Sales)
GROUP BY Customer
```

Figure 6-6. *Columnar database table with three projections*

C-Store storage (2)

- Data partitioning and distribution
  - Horizontally partitioned into 1 or more *segments*
  - Segments associate data values of every column with a *storage key*
  - *Join indices* reconstruct all the records in a table



C-Store storage (3)

- Columnar Compression
  - Usually try to work on localized subsets of the data
  - Data is stored in sorted order
  - Few “combinations of values” high compression is possible!
- Heavily compressed columns
  - Trade CPU time for I/O bandwidth!
  - CPU overhead of compression lower on isolated blocks of data (columns are stored together on disk)

C-Store optimizer and transactions

- A hybrid architecture
  - Frequent insert and update and query performance
  - Column-oriented optimizer and executor

- High availability and improved performance
  - K-safety using a sufficient number of overlapping projections

- The use of *snapshot isolation*, 2PC for writes
  - Timestamps for reads consistent in a given “snapshot”

Columns Stores hybrid architecture (1)

- *Writeable (Delta) Store*
  - Optimized for frequent writes
  - Like an in-memory row-oriented db
  - Periodically merged with the main columnar-oriented store by "*Tuple Mover*"

- *Read-Optimized Store*
  - *Main Column store*

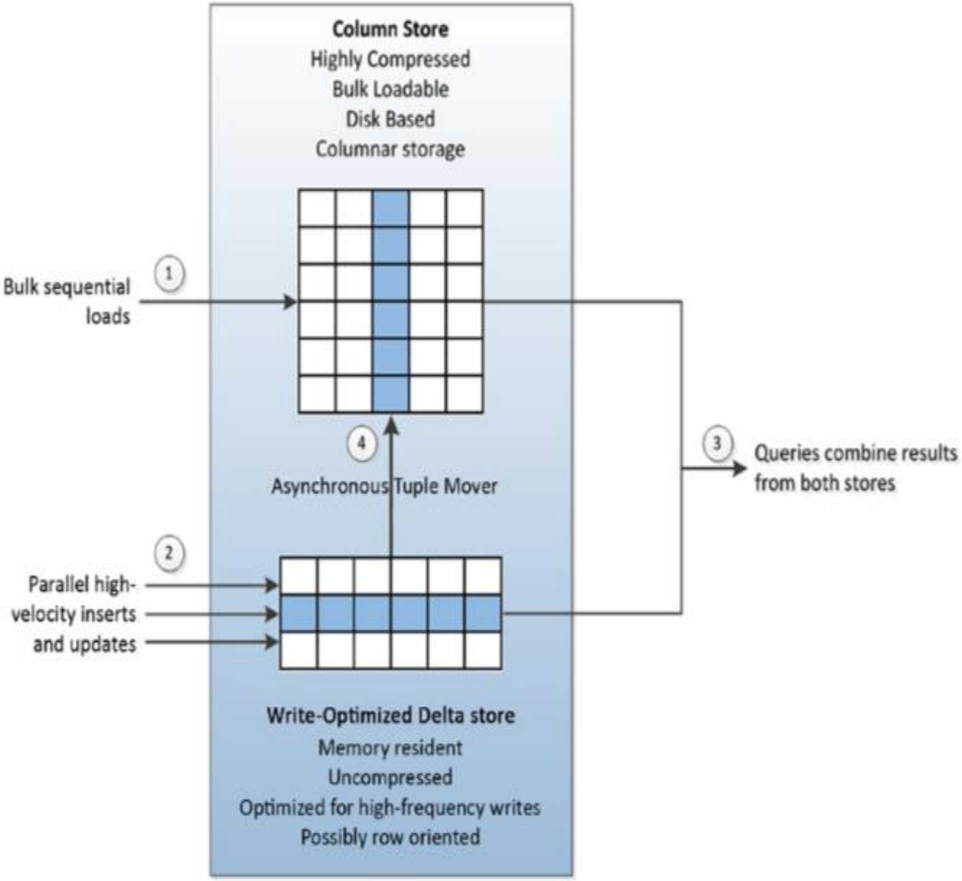


Figure 6-5. *Write optimization in column store databases*

Columns Stores hybrid architecture (2)

- Nightly ETL jobs
  - Column store
- Incremental inserts and updates
  - Write-optimized store
- Queries may need to read from both!

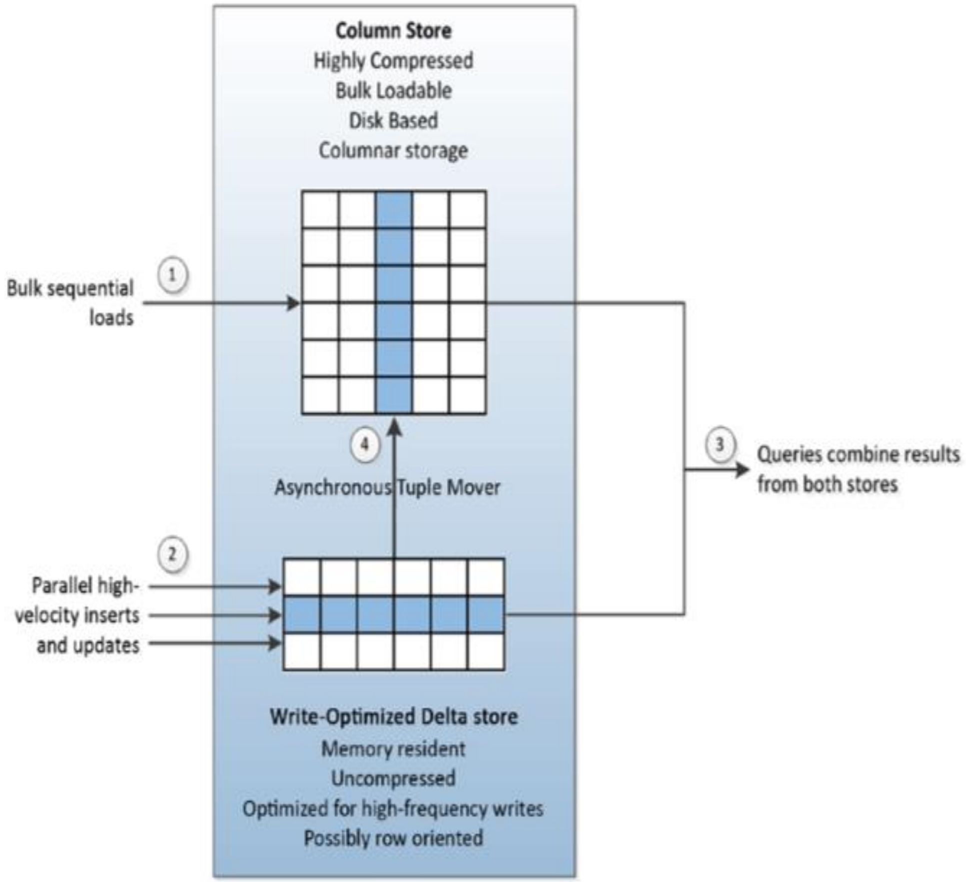


Figure 6-5. *Write optimization in column store databases*

Disks and Databases Architectures

- The magnetic disk device has been a pervasive presence within digital computing since the 1950s
  - Moore's law does not apply to the mechanical aspects of disk performance
  - But alternative technologies brought significant improvements!

- Solid state provide tremendously lower IO latencies
  - Performance of SSD is on orders of magnitude superior
  - Read operations require only a single-page IO
  - Writing a page requires an erase and overwrite of a complete block
  - *SSD is significantly slower in writes than in reads*

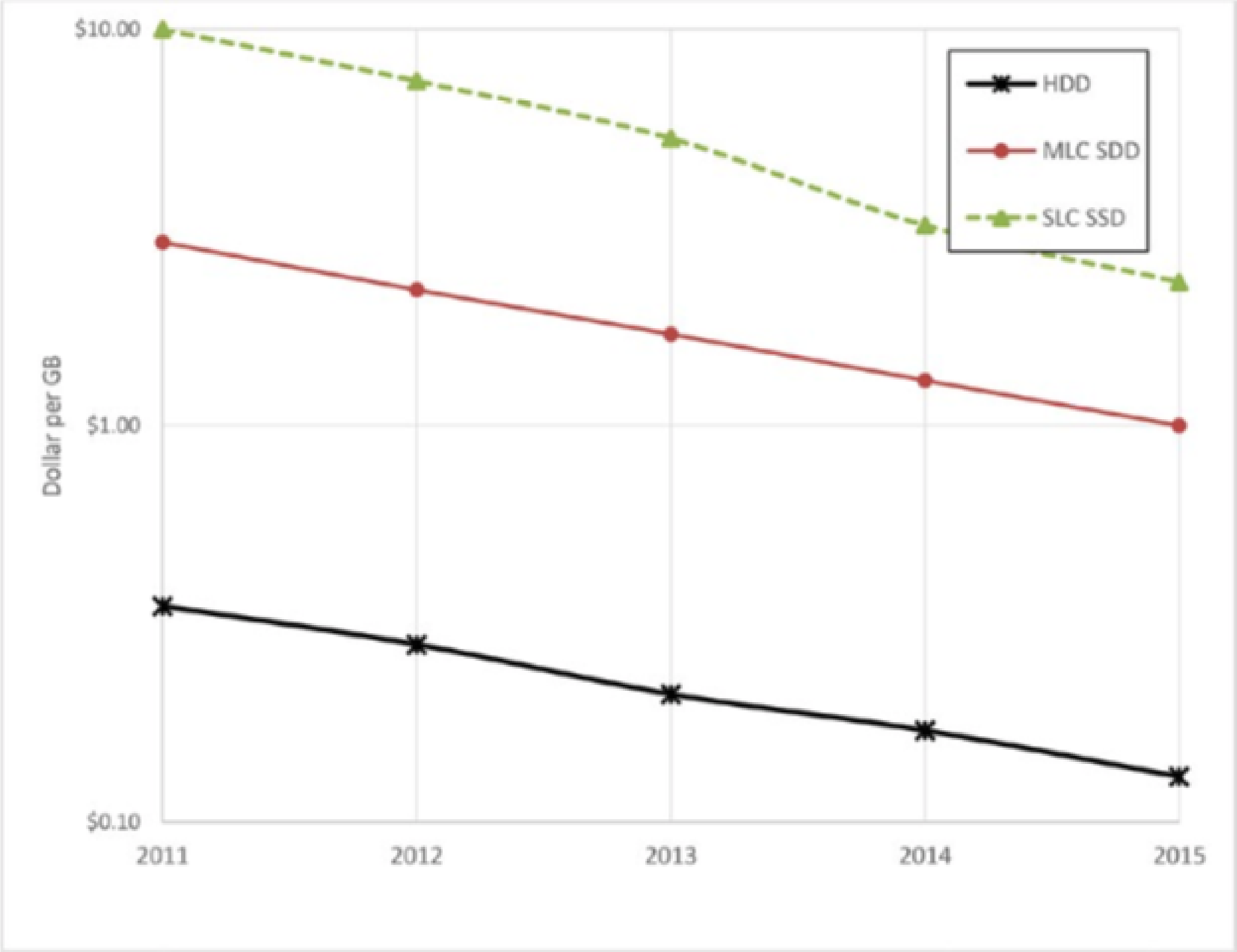


Figure 7-3. Trends in SSD and HDD storage costs (note logarithmic scale)

SSD and Databases

- Traditional relational databases perform relatively poorly on
  - Critical IO has been isolated to sequential write
  - The worst possible workload for SSD

- Some nonrelational systems
  - Avoid updating existing blocks and perform larger batched write operations (friendlier to solid state disk performance)

- *Aerospike – SSD-oriented*
  - NoSQL database
  - A log-structured file system
    - Updates are physically implemented by appending the new value to the file and marking the original data as invalid
  - Main memory to store indexes, data always on flash
  - “Avoid IO at all costs” approach unnecessary

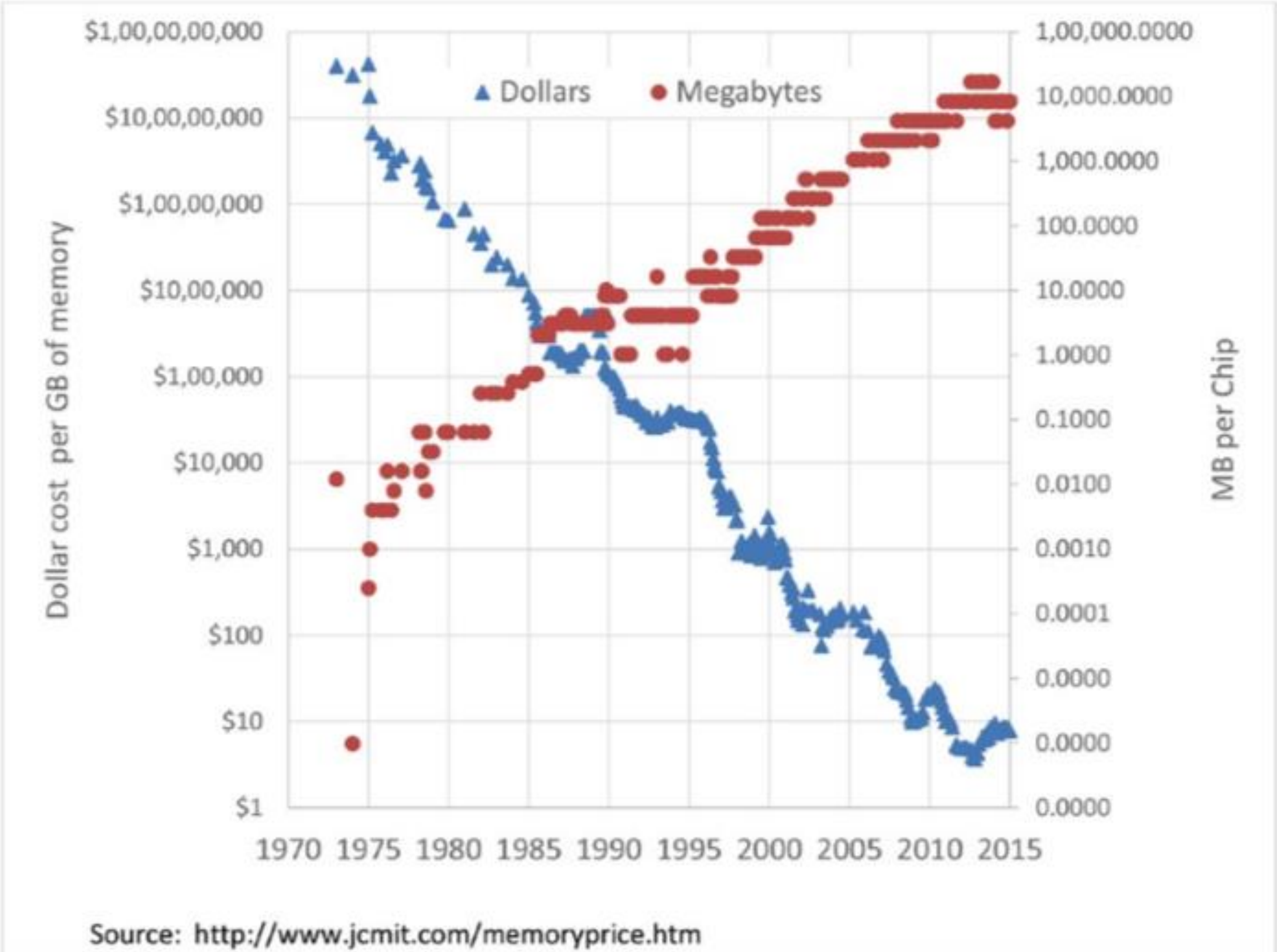


Figure 7-4. *Trends for memory cost and capacity for the last 40 years (note logarithmic scale)*

Main memory DB

- **Cache-less architecture**
  - Traditional disk-based databases cache data in main memory to minimize disk IO
  - There is no point caching in memory what is already stored in memory!

- **Alternative persistence model**
  - Data in memory disappears when the power is turned off
  - Alternative mechanism for ensuring no data loss
    - Replicating data to other members of a cluster
    - Writing complete database images to disk files
    - Writing out transaction/operation records to a *transaction log* or *journal*

TimesTen (1995 – oracle 2005)

- SQL-based relational model
- Data is memory resident
- Periodic snapshots of memory to disk
- Disk-based transaction log following a transaction commit

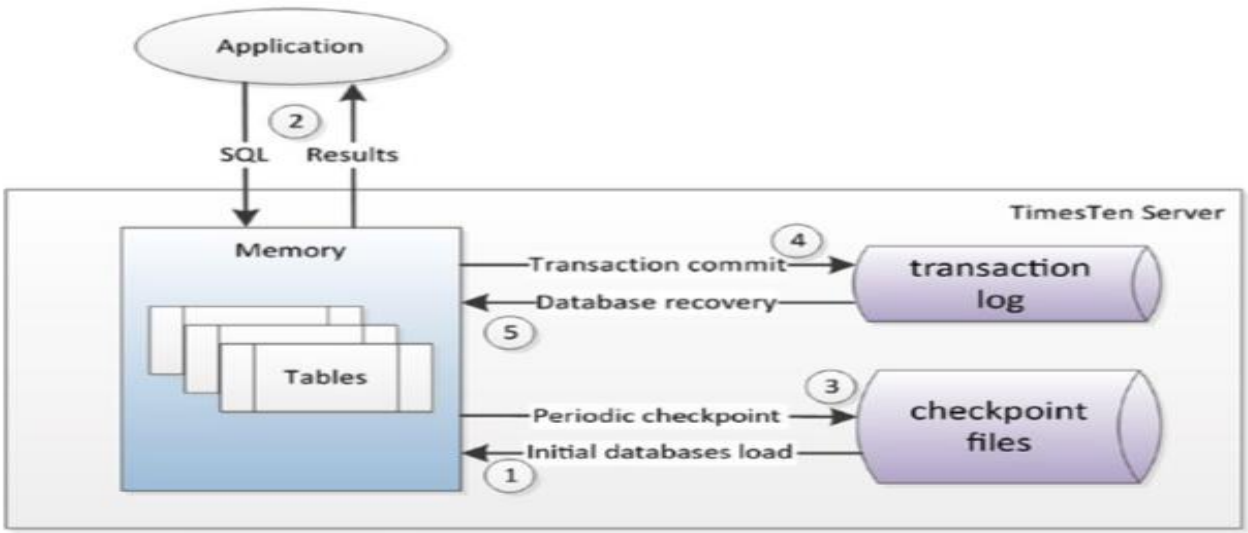


Figure 7-5. *TimesTen Architecture*

Redis (2009 ~ 2013 VMWare)

- Key-value store architecture

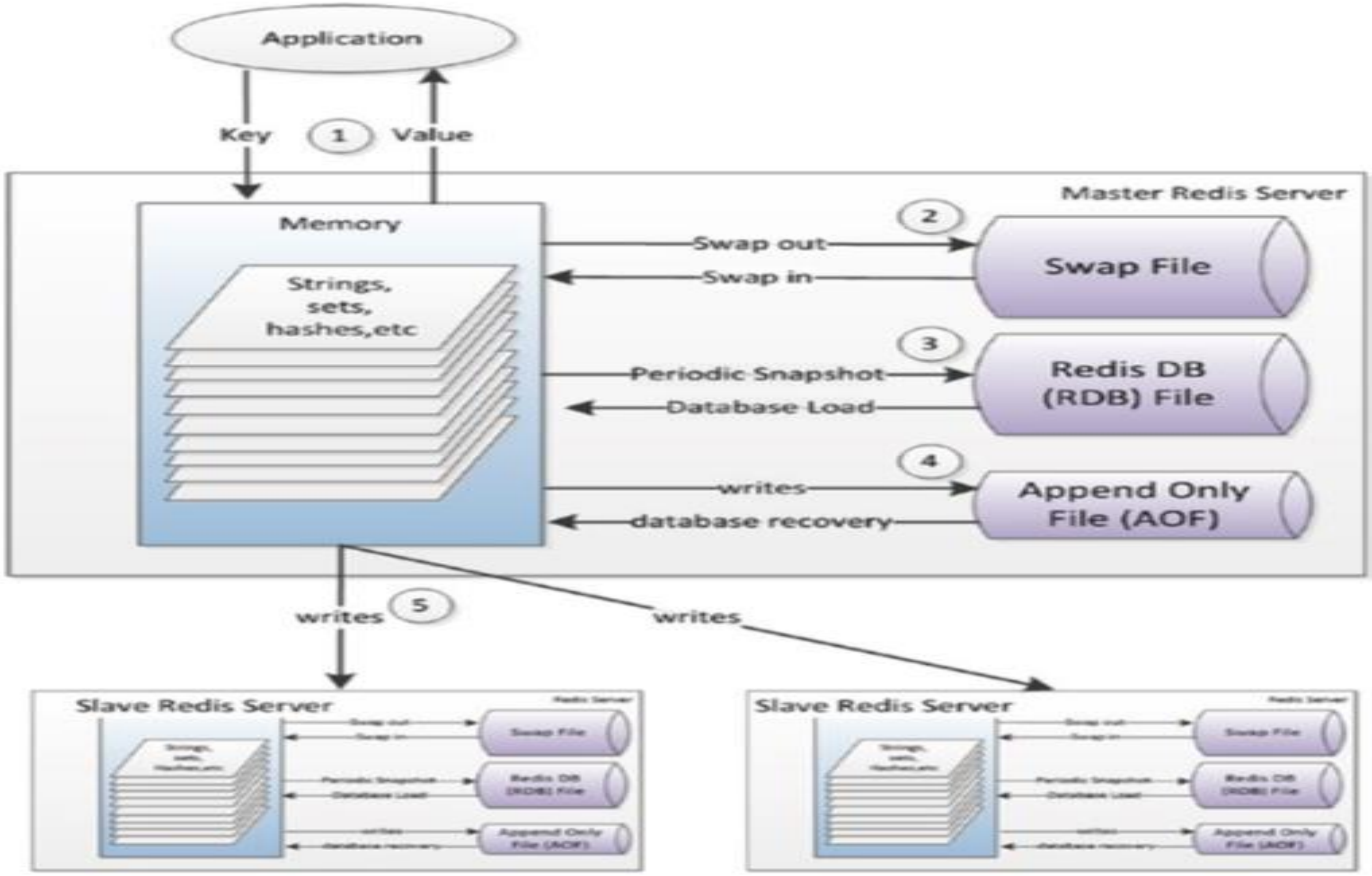


Figure 7-6. *Redis architecture*

SAP HANA

- In-memory database for Business Intelligence (BI)
- Capable of supporting OLTP workloads
- Column and Row Storage

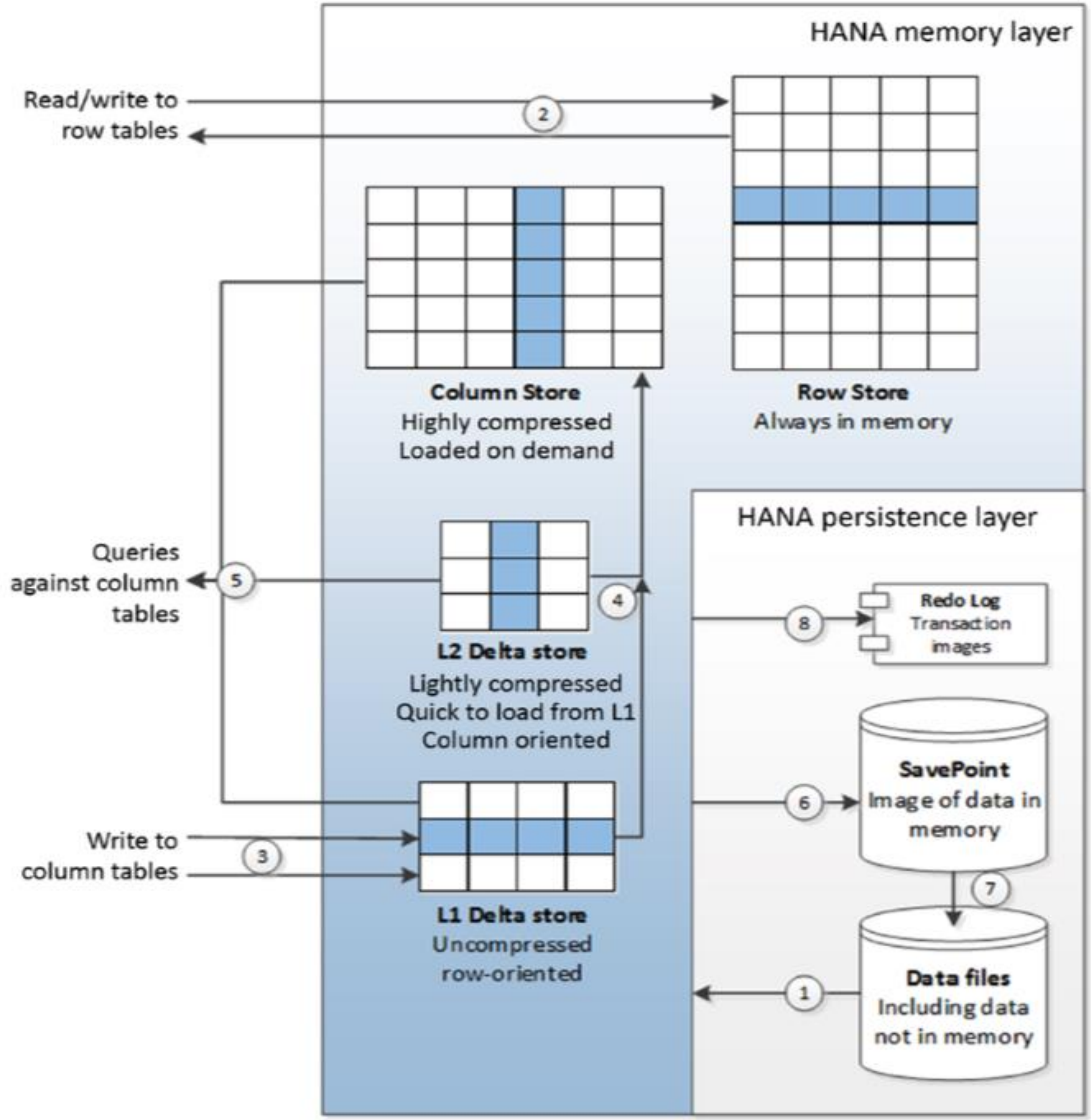


Figure 7-7. SAP HANA architecture

More in-memory databases

- VoltDB
  - Follows *H-store* design
  - A pure in-memory solution
  - Supports the ACID transactional model
  - Persistence through replication across multiple machines

- Oracle 12c “in-Memory Database”
  - In-memory column store to supplement its disk-based row store

Berkeley Analytics Data Stack (BDAS)

- **Spark**
  - In-memory, distributed, fault-tolerant processing framework.
  - Implemented in Scala
  - Higher-level abstractions than MapReduce
  - Excels at tasks that cause bottlenecks on disk IO in MapReduce
    - Good for tasks that iterate repeatedly over a dataset (e.g., machine-learning workloads)
- **Mesos**
  - Cluster management layer (analogous to Hadoop's YARN)
  - Intended to allow multiple frameworks
- **Tachyon**
  - Fault-tolerant, Hadoop-compatible, memory-centric distributed file system

Berkeley Analytics Data Stack and Spark

- In few words:
  - *Spark represents an in-memory variation on the Hadoop theme*
  - Made to cooperate with Hadoop, not to replace it!

- BDAS includes
  - *Spark streaming*
    - *Streams analytics*
  - *GraphX*
    - Graph computation based on Spark
  - *MLBase*
    - Machine-learning libraries at various levels of abstraction

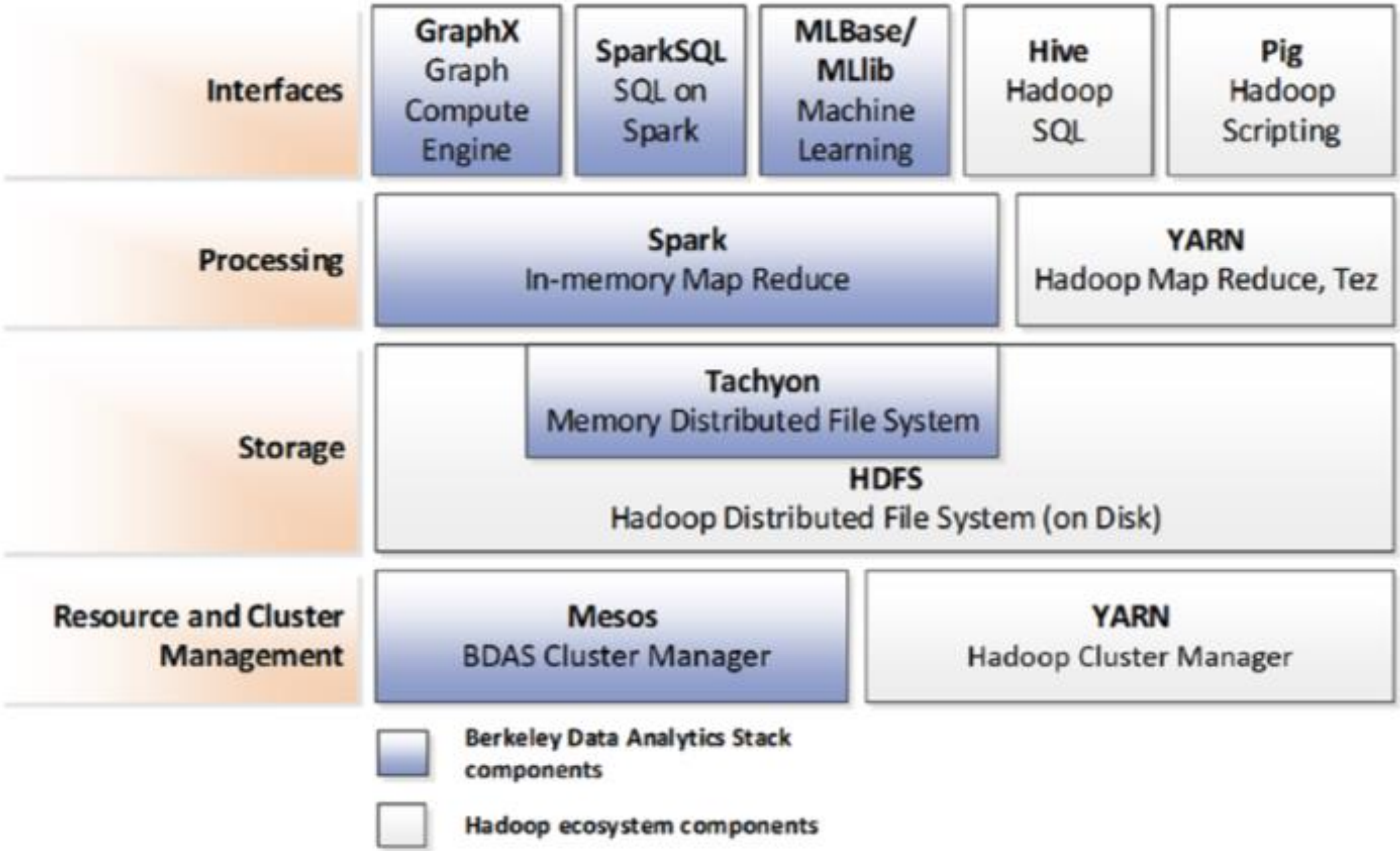


Figure 7-10. *Spark, Hadoop, and the Berkeley Data Analytics Stack*

Spark (1)

- Data as *resilient distributed datasets* (*RDD*)
  - Collections of objects
    - Simple types such as strings or any other Java/Scala object type
  - Partitioned automatically across nodes of the cluster
  - RDDs are immutable
    - Operations on RDDs return new RDDs
- Methods implemented by (*DAG*) operations
  - Directed acyclic graph
  - A more sophisticated paradigm than MapReduce
- Spark is not an OLTP-oriented system
  - No transaction log or journal

Spark (2)

- Manages data that won’t fit entirely into main memory
  - May page data to disk if the data volumes exceed memory capacity
- Can read from or write to local or distributed file systems
  - It integrates with Hadoop, works with HDFS or external data
  - RDDs may be represented on disk as text files or JSON documents
- Can access data in any JDBC-compliant database
- The Spark API defines high-level methods
  - Joining, filtering, and aggregation

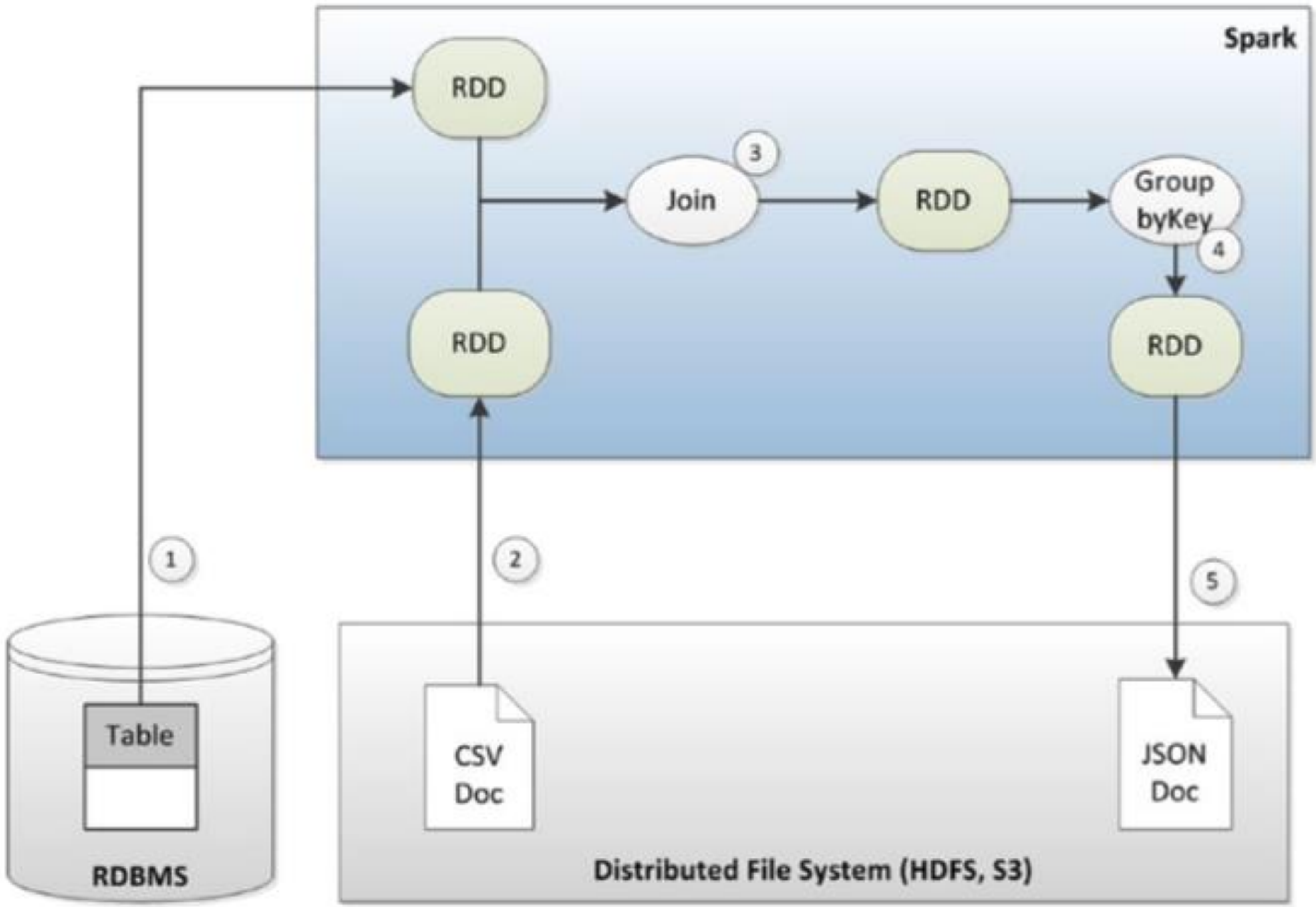


Figure 7-11. *Elements of Spark processing*