Advanced Data Management
Afternotes

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

[CM0520] ADVANCED DATA MANAGEMENT (CM90)
Academic Year 2022 - 2023

Student Zuliani Riccardo 875532

Contents

I Advanced Data Management Book **1**

1 Introduction **2**
1.1 Database Proprieties . . . 2
1.2 DB Design . . . 3

2 RDBMS **4**
2.1 Relational Data Model . . . 4
2.1.1 Database and Relation Schemas . . . 4
2.1.2 Mapping ER Models to Schemas . . . 7
2.1.3 Normalization . . . 7
2.2 Relational Query Languages . . . 8
2.2.1 Relational Algebra Operators . . . 8
2.3 Transaction . . . 9

3 New Requirements, NOSQL **10**
3.1 Weaknesses of the RDM . . . 10
3.2 Weaknesses of RDBMSs . . . 11
3.3 New Data Management Challenges . . . 11
3.4 NOSQL . . . 12

4 Key-value Stores **13**
4.1 Key-Value Storage . . . 13
4.1.1 Map-Reduce . . . 14

5 Column Stores **17**
5.1 Column-Wise Storage . . . 17
5.1.1 Column Compression . . . 18
5.1.2 Null Suppression . . . 22
5.2 Column striping . . . 22

6 Extensible Record Stores **24**
6.1 Logical Data Model . . . 24
6.2 Physical storage . . . 27
6.2.1 Memtables and immutable sorted data files . . . 28
6.2.2 File format . . . 30
6.3 Redo Logging . . . 32
6.3.1 Compaction . . . 33
6.3.2 Bloom filters . . . 35

7 Graph Databases **38**
7.1 Graphs and Graph Structures . . . 38
7.1.1 A Glimpse on Graph Theory . . . 38
7.1.2 Graph Traversal and Graph Problems . . . 40
7.2 Graph Data Structures . . . 40
7.2.1 Edge List . . . 41
7.2.2 Adjacency Matrix . . . 41
7.2.3 Incidence Matrix . . . 43
7.2.4 Adjacent List . . . 44
7.2.5 Incidence List . . . 45
7.3 The Property Graph Model . . . 47

8 Distributed Databases **50**
8.1 Scaling Horizontally . . . 50
8.2 Distribution Transparency . . . 51
8.3 Failures in Distributed Systems . . . 52
8.4 Epidemic Protocols and Gossip Communication . . . 53
8.4.1 Hash Trees . . . 55

9 Data Fragmentation **57**
9.1 Properties and Types of Fragmentation . . . 57
9.2 Data Allocation . . . 58
9.2.1 Consistent Hashing . . . 58

10 Replication And Synchronization **61**
10.1 Replication Models . . . 61
10.1.1 Master-Slave Replication . . . 62
10.1.2 Multi-Master Replication . . . 62

11 Polyglot DB Arch **64**
11.1 Polyglot Persistence . . . 64
11.2 Lambda Architecture . . . 65
11.3 Multi-Model Databases . . . 66

II Document Databases **68**

12 MongoDB **69**
12.1 Document . . . 69
12.1.1 Document Structure . . . 69
12.1.2 Dot Notation . . . 69
12.1.3 Document Limitations . . . 69
12.1.4 Other Uses of the Document Structure . . . 70
12.2 Databases and Collections . . . 70
12.2.1 Database . . . 70
12.2.2 Collections . . . 70
12.3 MongoDB Query API . . . 71
12.4 MongoDB CRUD Operations . . . 71
12.5 Aggregation Pipeline . . . 71

12.5.1 Aggregation Pipeline Expressions . . . 71
12.6 Replication . . . 72
12.6.1 Redundancy and Data Availability . . . 72
12.6.2 Replication in MongoDB . . . 72
12.6.3 Asynchronous Replication . . . 72
12.6.4 Automatic Failover . . . 73
12.7 Read Operations . . . 73
12.7.1 Data Visibility . . . 73
12.7.2 Mirrored Reads . . . 73
12.7.3 Transactions . . . 73
12.8 Sharding . . . 74
12.8.1 Sharded Cluster . . . 74
12.8.2 Shard Keys . . . 74
12.8.3 Advantages of Sharding . . . 75
12.8.4 Considerations Before Sharding . . . 75
12.8.5 Sharded and Non-Sharded Collections . . . 75
12.8.6 Connecting to a Sharded Cluster . . . 75
12.8.7 Sharding Strategy . . . 75
12.8.8 Zones in Sharded Clusters . . . 75
12.8.9 Transactions . . . 76

13 Cypher **77**
13.1 Cypher path matching . . . 77

III DBMS Internals **78**

14 DBMS Functionalities and Architecture **79**
14.1 Overview of a DBMS . . . 79
14.2 DBMS Architecture . . . 80
14.3 The JRS System . . . 81

15 Permanent Memory and Buffer Management **82**
15.1 Permanent Memory . . . 82
15.1.1 Parentheses on Magnetic Disk . . . 83
15.2 Permanent Memory Manager . . . 84
15.3 Buffer Manager . . . 85

16 Heap and Sequential Organizations **87**
16.1 Storing Collections of Records . . . 87
16.1.1 Record Structure . . . 87
16.1.2 Page Structure . . . 88
16.1.3 File Pages Management . . . 89
16.2 Cost Model . . . 90
16.3 Heap Organization . . . 90
16.3.1 Performance Evaluation - (NOT REQUIRED) . . . 90
16.4 Sequential Organization . . . 91
16.4.1 Performance Evaluation - (NOT REQUIRED) . . . 91
16.5 Comparison of Costs . . . 92

16.6 External Sorting 92
16.6.1 Performance Evaluation - (NOT REQUIRED) 93

17 Hashing Organization **95**
17.1 Table Organizations Based on a Key 95
17.2 Static Hashing Organization 96
17.2.1 Performance Evaluation 97
17.3 Dynamic Hashing Organizations 98
17.3.1 Virtual hashing 98
17.3.2 Linear Hashing 99

18 Dynamic Tree-Structure Organizations **101**
18.1 B-trees 101
18.1.1 Operations 102
18.2 Performance Evaluation 103
18.2.1 Search 103
18.2.2 Range Search 103
18.2.3 Insertion 103
18.2.4 Deletion 103
18.3 B^+-trees 104
18.4 Index Organization 104
18.4.1 Performance Evaluation - (NOT REQUIRED) 106

19 Non-Key Attribute Organizations **107**
19.1 Non-Key Attribute Search 107
19.2 Inverted Indexes 108
19.2.1 Performance Evaluation - (NOT REQUIRED) 108
19.2.2 Equality Search - (NOT REQUIRED) 109
19.2.3 Range Search - (NOT REQUIRED) 110
19.3 Bitmap indexes 110
19.3.1 Comparison with traditional indexes - (NOT REQUIRED) . . 111
19.4 Multi-attribute Index 111

20 Multidimensional Data Organizations **113**
20.1 Linearization- B^+-tree 114
20.1.1 Space partitioning 116
20.2 G-trees 118
20.2.1 Point Search 119
20.2.2 Spatial Range Search 119
20.2.3 Point Insertion 120
20.2.4 Point Deletion 120
20.3 R^*-trees 121
20.3.1 Search Overlapping Data Regions 121
20.3.2 Insertion 122

21 Access Methods Management **123**
21.1 The Storage Engine 123
21.2 Access Method Operators 123

22 Implementation of Relational Operator 125
22.1 Assumption and Notation . . . 125
22.1.1 Physical Data Organization . . . 125
22.1.2 Physical Query Plan Operators . . . 125
22.1.3 Physical Query Plan Execution . . . 125
22.2 Physical Operators for Relation R . . . 126
22.3 Physical Operator for Projection π^b . . . 127
22.4 Physical Operators for Duplicate Elimination δ . . . 127
22.5 Physical Operators for Selection σ . . . 128
22.6 Physical Operators for Grouping γ . . . 129
22.7 Physical Operators for Join $\bowtie$ . . . 129
22.8 Physical Operators for Set and Multiset Union, Intersection and Difference . . . 131

23 Query Optimization 132
23.1 Query Analysis Phase . . . 132
23.2 Query Transformation Phase . . . 132
23.2.1 DISTINCT Elimination . . . 133
23.2.2 GROUP BY Elimination . . . 133
23.2.3 WHERE-Subquery Elimination . . . 133
23.3 Physical Plan Generation Phase . . . 133
23.3.1 Single-Relation Queries . . . 134

24 Transaction And Recovery Management 135
24.1 Transaction . . . 135
24.1.1 Tansactions from the DBMS's Point of View . . . 136
24.2 Types of Failures . . . 136
24.3 Database System Model . . . 137
24.4 Data Protection . . . 137
24.5 Recovery Algorithms . . . 138
24.5.1 Use of the Undo Algorithm . . . 139
24.5.2 Use of the Redo Algorithm . . . 139
24.5.3 No use of the Undo and Redo Algorithms . . . 140
24.6 Recovery Manager Operations . . . 140
24.6.1 Undo-NoRedo Algorithm . . . 140
24.6.2 NoUndo-Redo Algorithm . . . 140
24.6.3 Undo-Redo Algorithm . . . 141
24.7 Recovery from System and Media Failures . . . 141

25 Concurrency Management 142
25.1 Introduction . . . 142
25.2 History . . . 142
25.3 Serializable History . . . 143
25.4 Serializability with Locking . . . 144
25.4.1 Strict Two-Phase Locking . . . 144
25.4.2 Deadlocks . . . 145
25.5 Serializability without Locking . . . 146
25.6 Multiple-Granularity Locking . . . 146
25.7 Locking for Dynamic Databases . . . 147

26 Distribute Concurrency Control **148**
26.1 Two-Phase Commit 148
26.2 Pasox Algorithm 150
26.3 Vector Clock 152
26.4 Version Vectors 152
26.5 Optimizations of Vector Clocks 154
26.5.1 Client IDs versus replica IDs 154

27 Consistency **155**
27.1 Strong Consistency 155
27.2 Weak Consistency 155
27.3 Write and Read Quorums 156

List of Figures

2.1 General example . . . 4
2.2 BookLending example . . . 5
2.3 Library schema example . . . 6
2.4 Library schema definition . . . 6
2.5 Book local dependency . . . 6
2.6 Library global dependencies . . . 6
2.7 Library schema normalized . . . 8

4.1 Example of Map Reduce . . . 14
4.2 Example of Combination Map Reduce . . . 16

5.1 BookLending example . . . 17
5.2 Run-length encoding example . . . 19
5.3 Bit vector encoding example . . . 19
5.4 Dictionary encoding example . . . 20
5.5 Dictionary sequence encoding example . . . 20
5.6 Frame reference encoding example . . . 20
5.7 Differential encoding example . . . 21

6.1 Column Families, Column Qualifiers, Values, Row Keys . . . 25
6.2 Ordering features of BookInfo . . . 26
6.3 Ordering features of LendingInfo . . . 26
6.4 Writing to memory tables and data files . . . 28
6.5 Reading to memory tables and data files . . . 28
6.6 File format of data files . . . 31
6.7 Write ahead log on disk . . . 32
6.8 Compaction on disk . . . 33
6.9 Level compaction . . . 34
6.10 Bloom filter . . . 37

7.2 A property graph for a social network . . . 48
7.3 Violation of uniqueness of edge labels . . . 49

8.1 A hash tree for four messages . . . 55

9.1 Data allocation with consistent hashing . . . 59
9.2 Server removal with consistent hashing . . . 59
9.3 Server addition with consistent hashing . . . 60
9.4 Virtual server with consistent hashing . . . 60

10.1 A Master-slave replication 62
10.2 Master-slave replication with multiple records 62
10.3 Multi-master replication 63

11.1 Polyglot persistence with integration layer 65
11.2 Lambda architecture 66
11.3 A multi-model database 66

14.1 Architecture of a DBMS 80

15.1 Characteristics of the three types of memory 83
15.2 The Buffer Manager 85

16.1 Representation of attribute values 88
16.2 Pointers to records 88
16.3 Sort-Merge 93
16.4 Example of Sort-Merge 93

17.1 Static hashing organization 96
17.2 Example of Virtual Hashing 99
17.3 Search operation 99
17.4 Example of Linear Hashng 100

18.1 B-tree example 101
18.2 Insertion key 70 - Step 1 102
18.3 Insertion key 70 - Step 2 102
18.4 A rotation example 103
18.5 B+tree Example 104
18.6 Types of indexes 104
18.7 Dense index 105
18.8 Sparse index 105

19.1 An Index on Quantity 107
19.2 An *inverted index* on Quantity 108
19.3 Shape of Φ function 110
19.4 A *bitmap index* on Quantity 111

20.1 Multidimensional representation of data on cities 113
20.2 Data on cities locations 113

Part I

Advanced Data Management Book

Chapter 1

Introduction

1.1 Database Proprieties

Database systems should guarantee a correct and reliable execution in several use cases. Database system fulfill the following properties:

- **Data management:**
  - A database system not only stores data, it also support operations for retrieval of data, search and updates.
  - Database system should also support transactions: a sequence of operations on data in a database that must not be interrupted, or in others words either all operations succeed to their full extent or none of the operations is executed.
- **Scalability:** processing data by its distribution in a network of database servers and a high level of parallelization.
- **Heterogeneity:** when collecting data or producing data, could be saved in different type of structure:
  - *Structured:* if the data is in relational format which describe a fixed schema
  - *Semi-structured:* like tree or more in general graphs
  - *Unstructuired:* text documents
- **Efficiency:** the majority of database applications need fast database systems, so fast storage and retrieval.
- **Persistence:** providing a long-term storage facility for data, implementing also recovery system like transaction manager to allow to go back after a crash or continue if the operation has finish correctly. (Transaction could take some time)
- **Reliability:** database must prevent data loss and they have to support some kind of recovery system, thus they have to ensure:
  - *Data integrity:* data stored in the database system should not be distorted unintentionally

  - *Physical redundancy:* storing copies of data on other servers

- **Consistency:**
  - Database system must do its best to ensure that no incorrect or contradictory data persist in the system.
  - Automatic verification of consistency constraints
  - Automatic update of distributed data copies

- **Non-redundancy:**
  - Duplication of values inside the stored data sets should be avoided
  - Data sets with logical redundancy are prone to different forms of anomalies
  - *Normalization* is one way to transform data sets into a non-redundant format

- **Multi-User Support:** support concurrent accesses by multiple users or applications, with the constraint of isolation and not interference between users. A major issue is the control access: in which data must be protected like the usage of *views* or implementing an *authentication mechanism*

1.2 DB Design

The design phase should clearly answer basic questions like:

- Which data are relevant for the customers or the external applications?
- How should these relevant data be stored in the database?
- Which are usual access patterns on the stored data?

For conventional database systems changing the schema on a running database system is complex and costly, thus database design should be done with due care and following design criteria like the following:

- **Completeness:** all aspects of the information needed by the accessing applications should be covered
- **Soundness:** all information aspects and relationships between different aspects should be modeled correctly
- **Minimality:** no unnecessary or logically redundant information should be stored
- **Readability:** no complex encoding should be used
- **Modifiability:** changes in the structure of the stored data are likely to occur when running a database system over a long time
- **Modularity:** the entire data set should be divided into subsets

There are several graphical languages for database design, like Entity-Relationship Model (ERM) and the Unified Modeling Language (UML). For further information refer to the book.

Chapter 2

RDBMS

The relational data model is based on the concept of storing records of data as rows inside tables.

- Each **row** represents an entity of the real world
- And each **column** is an attribute of interest of these entities

2.1 Relational Data Model

2.1.1 Database and Relation Schemas

A relational database consists of a set of tables. Each table has

- Predefined name: **relation symbol**
- Set of predefined column names **the attribute names**
  - Each attribute A_i ranges over a predefined domain $dom(A_i)$ such that the values in the column can only come from this domain.

A table is then filled row-wise like a tuple with values that represent the state of an entity.

The definition of the attribute names A_i for the relation symbol R is called a **relation schema**; the set of the relation schemas of all relation symbols in the database is then called a **database schema**.

Relation Symbol R	Attribute A_1	Attribute A_2	Attribute A_3
Tuple $t_1 \rightarrow$	value	value	value
Tuple $t_2 \rightarrow$	value	value	value

Figure 2.1: General example

BookLending	BookID	ReaderID	ReturnDate
	123	225	25-10-2016
	234	347	31-10-2016

Domains:
$dom(BookID)$ and $dom(ReaderID)$: Integer
$dom(ReturnDate)$: Date

Figure 2.2: BookLending example

In addition each relation schema can have some type of *constraints* to describe which dependencies between the stored data exist:

- **INTRA relational constraints (local dependencies):** describe dependencies inside a single table, for example be *functional dependencies* – and in particular **key** constraints
- **INTER relational constraints (global dependencies):** describe dependencies between different tables, for example be *inclusion dependencies* – and in particular *foreign keyindex* constraints

We define a **Relational Schema** given its

- Relation Symbol R_i
- Set of attributes $A_{i1}, ..., A_{im}$
- Set of intrarelational/local constraints Σ_i

$$R_i = (\{A_{i1}, ..., A_{im}\}, \Sigma_i)$$

We define a **Database Schema** given its

- Database Symbol D
- Set of relation schemas $R_1, ..., R_m$
- Set of interrelational/global dependencies Σ

$$R_i = (\{A_{i1}, ..., A_{im}\}, \Sigma_i)$$

Example Database: Library

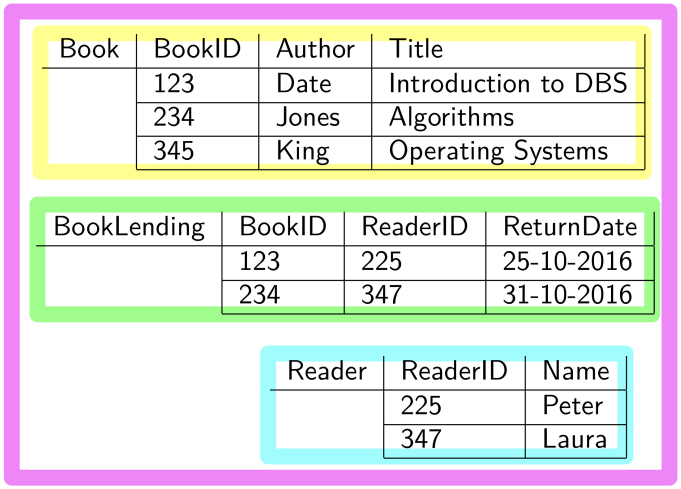


Figure 2.3: Library schema example

Relation schema 1: Book=({BookID, Author, Title},Σ_{Book})
Relation schema 2: BookLending=({BookID, ReaderID, ReturnDate},$\Sigma_{BookLending}$)
Relation schema 3: Reader=({ReaderID, Name},Σ_{Reader})
Database schema: Library = ({Book, BookLending, Reader},Σ)

Figure 2.4: Library schema definition

Example Database: Dependencies

Book	BookID	Author	Title
	123	Date	Introduction to DBS
	234	Jones	Algorithms
	345	King	Operating Systems

Figure 2.5: Book local dependency

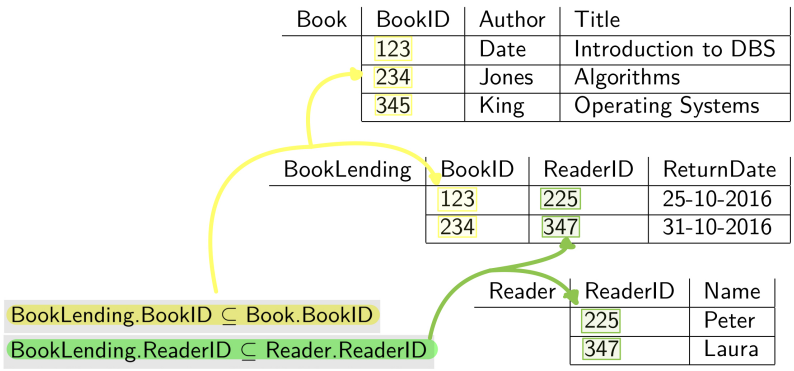


Figure 2.6: Library global dependencies

2.1.2 Mapping ER Models to Schemas

An ER diagram can be translated into a database schema:

- Each **entity name** corresponds to a relation symbol
- Each **entity attributes** correspond to relation attributes
  - Relational data model does not allow multi-valued ad composite attributes
  - In the case of multi-valued attributes, a new relation schema for each multi-valued attribute created containing additional foreign keys
- **Composite attributes** should usually be treated as single-valued attributes
- **Relationship** are also translated into a relation schema. In order to be able to map the values from the entities connected by the relationship together, the relation also contains the key attributes of the entities participating in the relationship

2.1.3 Normalization

Some database designs are problematic, they contain too many attributes, or tables combine the "wrong" attributes, or tables store data duplicates. These problem cause problem when *inserting*, *deleting* or *updating values*: they are called **anomalies**, and there are many types like:

- **Insertion anomaly:** we need all attribute values before inserting a tuple
- **Deletion anomaly:** when deleting a tuple, information is lost that we still need in the database
- **Update anomaly:** when data are stored redundantly, values have to be changed in more than one tuple

> Normalization results in a good distribution of the attributes among the tables and hence normalization helps reduce anomalies. Moreover it depends on the functional dependencies of the relative data table.

We have seen 3 main types of Normal form:

1. **First Normal Form (1NF)** that erase the multi-valued and allow composite attributes
2. **Second Normal Form (2NF)** in which all non-key attributes are fully dependent on the key attributes
3. **Third Normal From (3NF)** in which all non-key attributes are directly dependent on the key attributes

Example of Normalization

Book	BookID	Title
	1002	Introduction to DBS
	1004	Algorithms
	1006	Operating Systems

Reader	ReaderID	Name
	205	Peter
	207	Laura

BookLending	BookID	ReaderID	ReturnDate
	1002	205	25-10-2016
	1006	205	27-10-2016
	1004	207	31-10-2016

Figure 2.7: Library schema normalized

2.2 Relational Query Languages

After having design the relational database the next point is to define a strategy to retrieve some information, update and insert additional data in our just born database. We can choose on different strategy, for example taking into account the previous Library example, there is:

- **Natural Language:** "Find all lent books that must be returned before 29-10-2016"
- **Relational Calculus:** Logical formula with variables x, y, z :

$$Q(x, y, z) = \{(x, y, z)|\text{BookLending}(x, y, z) \ \wedge z < \text{29-10-2016}\}$$

- **Relational Algebra** $\sigma_{ReturnDate<29-10-2016}(BookLending)$

2.2.1 Relational Algebra Operators

- **Projection** π to make some attribute restriction. IDs of readers currently reading a book: $\pi_{ReaderID}(BookLending)$
- **Selection** σ with condition on answer tuples. All books to be returned before 29-10-2016: $\sigma_{ReturnDate<29-10-2016}(BookLending)$
- **Renaming** ρ to assign a new attribute name. Rename ReturnDate to DueDate $\rho_{DueDate \leftarrow ReturnDate}(BookLending)$
- **Union, Difference and Intersection**
- **Natural Join** $\bowtie$ to combine two tables on attributes with same name. All information on Books currently lent: Book $\bowtie$ BookLending

One more thing to note about relational algebra is that an algebra query can be illustrated by a tree:

- *Inner nodes* are the algebra operators
- *Leaf nodes* are the relation symbol

2.3 Transaction

A transaction can be characterized as a sequence of read and write operations on a database; this sequence of operations must be treated as a whole and cannot be interrupted. A transaction moves our system state from one into an other. The following propriety are ensured:

- **Logical data integrity:** are the written values correct and final results of a computation?
- **Physical data integrity & Recovery:** how can correct values be restored after a system crash?
- **Multi-user support:** how can users concurrently operate on the same database without interfering?

To achieve physical data integrity and as part of the recovery management, the database system maintains a **transaction log**. It stores which operation of which transaction is currently being executed. After a system restart all operations of uncommitted transactions have to be undone. The transaction log also has to take care of committed transactions: If all results of a transaction have been computed but disk writing is interrupted, after a system restart all affected computations of the transaction have to be redone and then stored to disk.

Most RDBMS manage transactions according to the following properties:

- **A**tomicity: either execute all operations or none of them
- **C**onsistency: after the transaction, all values in the database are correct
- **I**solation: concurrent transactions of different users do not interfere
- **D**urability: all transaction results persist in the database even after a system crash

Chapter 3

New Requirements, NOSQL

3.1 Weaknesses of the RDM

The following weaknesses can become an issue when using the relational data model

- **Inadequate Representation of Data:**
  - Translating arbitrary data into the *relational table* format is not an easy task, since there are complex formats like XML and unstructured text documents, so squeezing data into row and columns need some additional attention
  - The efficiency of retrieval is associated also to the data representation since it might have to be recombined from several tables
  - Due to normalization data belonging to a single entity might end up in several different tables
- **Semantic Overloading:** entities as well as relationships between entities are both mapped to relations in a database schema. In our final database schema, both the entities and the relationships were contained as relation schemas.
- **Weak Support for Recursion:** in the relational data model it is diffcult to execute recursive queries that need to join. The purpose of a recursive query is to compute the *transitive closure* of some table attributes. Costly to compute in a real RDBMS.
- **Homogeneous data structure:** the relational data model requires both *horizontal* and *vertical* homogeneity:
  - *Horizontal homogeneity:* all tuples have to range over the same set of attributes
  - *Vertical homogeneity:* values in one column have to come from the same predefined attribute domain

  Mixing values from different domains in one column is not allowed

3.2 Weaknesses of RDBMSs

- **Infrequent updates:** RDBMSs are designed for frequent queries but very infrequent updates
- **SQL dialects:** not all RDBMSs fully support the standard and some deliberately use their own syntax
- **Restricted data types:** RDBMS can be considered quite inflexible regarding the support of modern data types or formats
- **Declarative access:** SQL is that queries are usually *declaratively* expressed based on the (expected) content in the database tables. However, other data formats might require a non-declarative access
- **Short-lived transactions:** the typical RDBMS transaction is however very short-lived. The implemented transaction management mechanisms are usually not suited for long-term transactions. However, the support for long-term transactions is in particular important for data stream processing where queries are periodically executed on continuous streams of data
- **Lower throughput:** handling massive amounts of data, achieving a sufficiently high data throughput might not be possible as good as one would require with an RDBMS
- **Rigid schema:** schema evolution is poorly supported: Changes in the relation schemas are diffcult and costly
- **Non-versioned data:** versioning of data is disregarded by conventional RDBMSs

3.3 New Data Management Challenges

Some of the new challenges for database management are the following:

- **Complexity:** data are organized in complex structures like social network and thus graph structure
- **Schema independence:** documents can be processed without a given schema definition, so data can be structured in an arbitrary way without complying with any prescribed format
- **Sparseness:** if there is an (optional) schema for a data set, it may happen that a lot of data items are not available
- **Self-descriptiveness:** As a consequence on schema independence and sparseness, metadata are attached to individual values in order to enable data processing
- **Variability:** data are *constantly changing*, DBMS has to handle frequent data modifications in the form of insertions, updates and deletions

- **Scalability:** data are distributed on a huge number of interconnected servers. Moreover, the database system has to support flexible horizontal scaling : servers can leave the network and new servers can enter the network on demand.
- **Volume:** large data volumes have to be processed

3.4 NOSQL

Non-relational databases have been developed as a reaction to these challenges and new requirements. Historically the term "NoSQL" applied to database systems that offered query languages and access methods other than the standard SQL. NOSQL covers database systems that:

- Have data models other than the conventional relational tables
- Support programmatic access to the database system or query languages other than SQL
- Can cope with schema evolution
- Support data distribution
- Do not strictly adhere to the ACID properties

Chapter 4

Key-value Stores

Key-value stores are specialized for the efficient storage of simple key value pairs. Parallel processing of key-value pairs has been popularized with the **Map-Reduce paradigm**.

4.1 Key-Value Storage

A key-value pair is a tuple of two strings $\langle key, value \rangle$.

- The key is the identifier and has to be unique.
- You can retrieve a value from the store by simply specifying the key; and you can delete a key-value pair by specifying the key.
- A key-value store is the prototype of a **schemaless** database system: you can put arbitrary key-value pairs into the store and no restrictions are enforced on the format or structure of the value.

Key-value store basically only offers three operations:

- *reading:* value = store.get(key)
- *writing:* store.put(key, value)
- *deleting:* store.delete(key)

It is **"simple but quick"**, indeed data are stored in a simple key-value structure and the key-value store is ignorant of the content of the value part. Here some characteristics:

- Simple format is that data can easily be distributed among several database servers
- Key-value stores are good for "data-intensive" applications, like session management or shopping carts
- Most key-value stores, values are allowed to have other data types than just strings
- Some key-value stores also support data formats like XML or JSON

4.1.1 Map-Reduce

- **Map:** transform a given list into an other with all the elements passed to a given function
- **Reduce:** given a list and a function they produce a single value using all list elements

The basic elements of map-reduce are four functions that operate on key-value pairs split, map, shuffle and reduce. While *split* and *shuffle* are more or less generic functions that can have the same implementation for all applications, the other two – *map* and *reduce* – are **highly application-dependent** and have to be implemented by the user of the map-reduce framework.

Map and reduce are executed by *several worker processes* running on several servers; one of the workers is the **master** who *assigns* new map or reduce tasks to idle workers.

- **Split** input key-value pairs into disjunct subsets and assign each subset to a worker process
- Let workers compute the **map** function on each of its input splits that outputs intermediate key-value pairs
- The **shuffle** groups all intermediate values by key and assign each group to a worker
- Finally **reduce** values of each group and return the result

Example, counting occurrences of words in a document

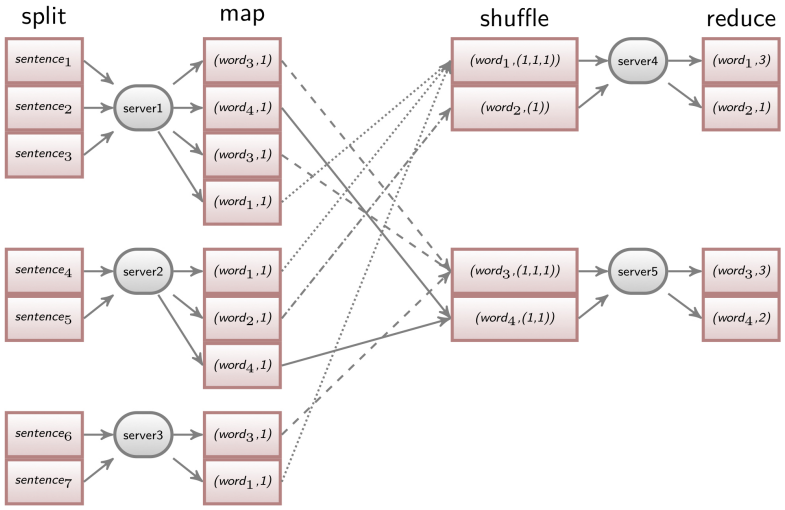


Figure 4.1: Example of Map Reduce

1. **Split function** splits the document into sentences; each sentence is assigned to a worker process
2. Worker thread starts a **map function** for each sentence.
   - It *parses* a sentence and for *each word* $word_i$, the worker thread emits a *key-value pair* $(word_i, 1)$ to indicate it encountered $word_i$ once
   - Intermediate result stored locally
3. In the **suffle phase** local intermediate results are read and grouped by words
   - The 1-values for each word are concatenated into a list
   - Key-value pair where the word $word_i$ is the **key** and the **value** is a *list of 1s* corresponding to individual occurrences of the word in all sentences
   - Each word is assigned to a worker process
4. The **reduce function** calculates the total number of occurrences by *summing the 1s*. The final results will look like a sequence of key-value pair $(word_i, sum_i)$

More formally:

- **Split:** *list ! list*$(key_1, value_1)$: split maps some input text to a list of key-value pairs
- **Map:** $(key_1, value_1)$! $list(key_2, value_2)$: map processes one key-value pair and maps it to a list of key-value pairs
- **Shuffle:** $list(key_2, value_2)$! $(key_2, list(value_2))$: shuffle groups the individual key-value pairs by key and appends to each key a list that is a concatenation of the values of the individual pairs
- **Reduce:** $(key_2, list(value_2))$! $(key_3, value_3)$: reduce aggregates a list of values into a single one

Possible optimizations

- **Parallelization:** the map and as well as the reduce task can be run in parallel by different concurrent worker processes and even on multiple servers
- **Partitioning:** usually there are more reduce tasks to be executed than workers available. That is, each worker has to execute several reduce task on a set of different keys

- **Combination:** instead of locally storing lots of intermediate results of the map processes which later on have to be shuffled to other workers over the network. *Combine* task can be run locally on each worker after the map phase, it is similar to the reduce function as it groups the intermediate results by key and combines their values

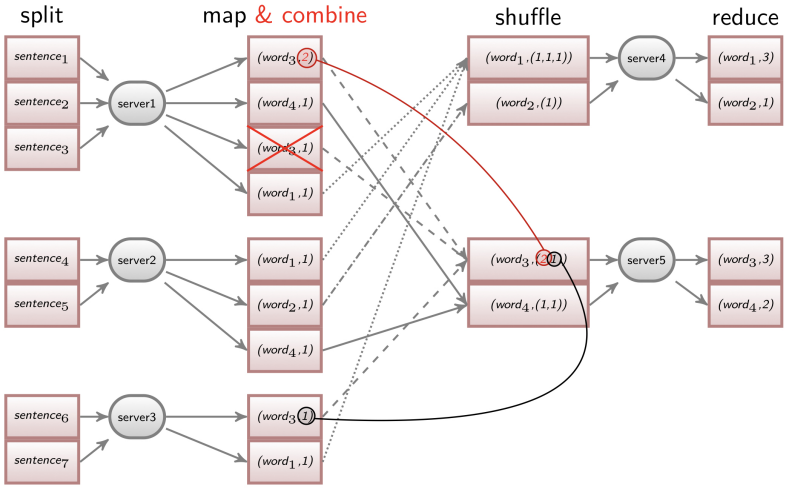


Figure 4.2: Example of Combination Map Reduce

- **Data Locality:** transmitting data to a worker over the network is costly, thus the *master* can take the location of data into account before assigning a task to a worker

- **Incremental Map-Reduce:** input data might be generated dynamically over a longer period of time. To improve evaluation of such data, the four steps can be interleaved and final results be obtained incrementally

Chapter 5

Column Stores

Row Store:

- Data are stored in tables and on disk
- Stored consecutively
- Used in most RDBMs

Column Store:

- Column oriented relational database
- Data are stored in tables but on disk
- Stored consecutively

Column store work very well for certain queries type like queries that can be executed on compressed data, indeed columns stores have the advantage of both a *compact storage* as well as *efficient query execution*.

5.1 Column-Wise Storage

We take up our tiny library example to illustrate the differences:

BookLending	BookID	ReaderID	ReturnDate
	1002	205	25-10-2016
	1006	207	31-10-2016

Figure 5.1: BookLending example

Storage order in row store:
1002, 205, 25-10-2016, 1006, 207, 31-10-2016
Storage order in column store:
1002, 1006, 205, 207, 25-10-2016, 31-10-2016
Advantages of column stores are for example:

- **Buffer management:** only columns (attributes) that are needed are read from disk into main memory, because a single memory page ideally contains all values of a column.

  - In row stores memory page might contain also other attributes, so data is fetched unnecessarily

- **Homogeneity:** values in a column have the same type. This is why they can be compressed better when stored consecutively.
  - In row stores values from different attribute domains are mixed in a row, so compression is not as good

- **Data locality:** iterating or aggregating over values in a column can be done quickly.
  - In row stores values have to be read and picked out from different tuples

- **Column insertion:** adding new columns to a table is easy because they just can be appended to the existing ones
  - In row store, storage reorganization it necessary to append a new column value to each tuple

Disadvantages of column stores are:

- **Tuple reconstruction:** combining values from several columns is costly because "tuple reconstruction" has to be performed
  - In row store, tuple reconstruction is not necessary because values of a tuple are stored consecutively

- **Tuple insertion:** inserting a new tuple is costly
  - In row store, the new tuple is appended to the existing ones and the new values are stored consecutively

5.1.1 Column Compression

- Values in a column range over the same attribute domain; that is, the have the same data type.
- Columns may contain lots of repetitions of individual values or sequences of values.
- These are two reasons why compression can be more effective on columns.
- Storage space needed in a column store may be less than storage space needed in a row store with the same data.

Having said that we can introduce five option for column data compression:

- **Run-length encoding:**
  - The run-length of a value denotes how many repetitions of the value are stored consecutively
  - We will store the value together with its starting row and its run-length
  - This encoding is most efficient for long runs of repetitive values

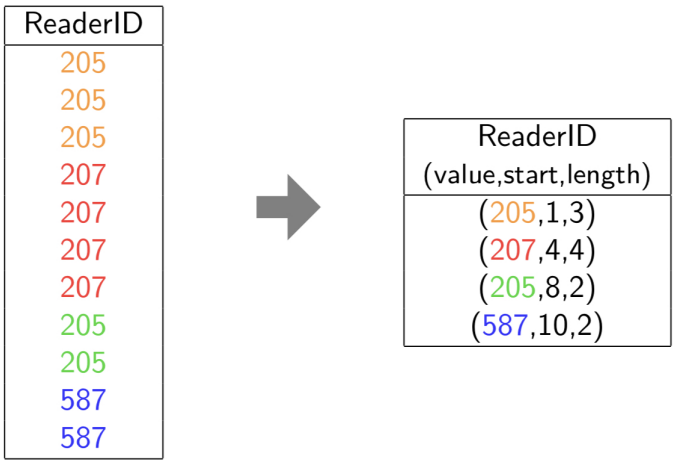


Figure 5.2: Run-length encoding example

- **Bit-vector encoding:**
  - For each value in the column we create a bit vector with one bit for each row
  - In each vector cell if the value is 1, it means the presence of the value in the column
  - This encoding is most efficient for relatively few distinct values and hence relatively few bit vectors

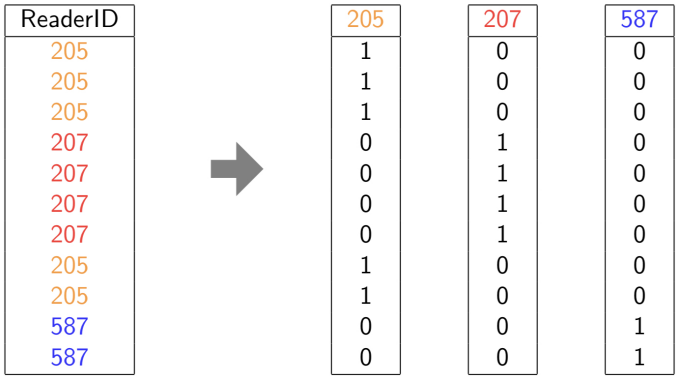


Figure 5.3: Bit vector encoding example

- **Dictionary encoding:**
  - We replace long values by shorter placeholders and maintain a dictionary to map the placeholders back to the original values

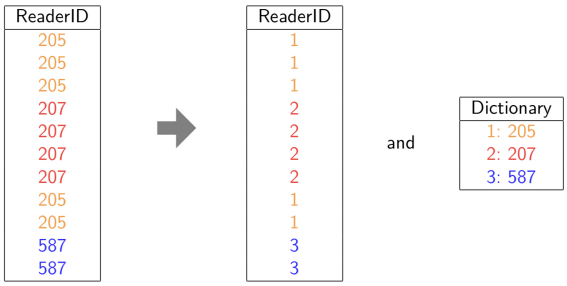


Figure 5.4: Dictionary encoding example

  - In some cases, we could not only create a dictionary for single values but even for frequent sequences of values

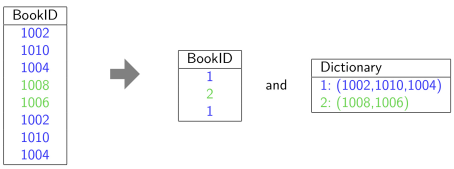


Figure 5.5: Dictionary sequence encoding example

- **Frame of reference encoding:**
  - For the range of values stored in a column, one value that lies in the middle of this range is chosen as the reference value
  - For all other values we only store the off-set from the reference value, (smaller than the original value)
  - 3 bit to store all. Interval $(-4, 4)$ so 2 bit plus the sign bit

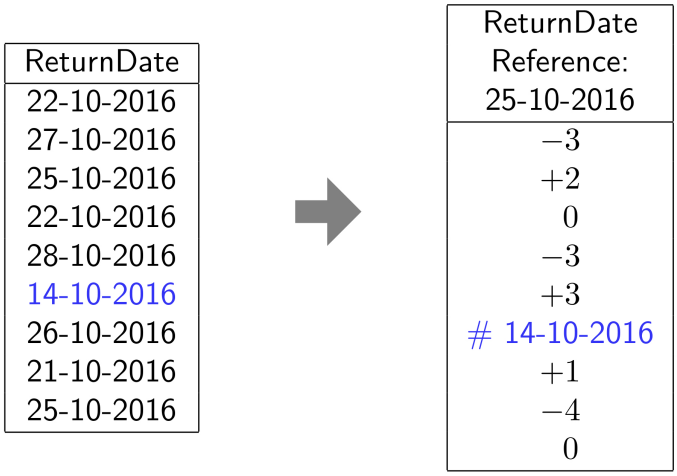


Figure 5.6: Frame reference encoding example

- **Differential encoding:**
  - An offset is stored instead of the entire value
  - The offset is the difference between the value itself and the value in the preceding row
  - Again the offset should not exceed a fixed size
  - As soon as the offset gets too large, the current value is stored as an *exception*
  - This encoding is only applicable to numerical data
  - It works best if data are stored in a sorted way
  - In addition with each encoding we have to maintain the order inside the column as otherwise tuple reconstruction would be impossible

ReturnDate
22-10-2016
27-10-2016
25-10-2016
22-10-2016
28-10-2016
14-10-2016
26-10-2016
21-10-2016
25-10-2016

ReturnDate
22-10-2016
+5
−2
−3
+6
14-10-2016
−2
−5
+4

Figure 5.7: Differential encoding example

  - **Disadvantages:**
    - Extra runtime needed to compute the encoding
    - Extra runtime needed to decode the data whenever we want to execute a query on them
  - Fortunately some queries can actually be executed on the compressed data so that no decompression step is needed. Like the following example: We have that the ReaderID is encoded as a sequence as follows:

((205, 1, 3), (207, 4, 2), (205, 6, 1))

Query: "How many books does each reader have?"

SELECT ReaderID, COUNT(*) FROM BookLending GROUP BY ReaderID

Now the column store does not have to decompress the column into the entire original column with 6 rows. It just returns (the sum of) the run-lengths for each ReaderID value.

- *Reader 205* is $3 + 1 = 4$
- *Reader 207* is 2

Result: {(205, 4), (207, 2)}

5.1.2 Null Suppression

Sparse columns are columns that contain many NULL values. A more compact format of a sparse column can be achieved by **not storing these null values**, but some additional information is needed to distinguish the non-null columns from the null columns

- **Position list:**
  - List that stores only the non-null positions but discards any null values
  - As metadata the total number of rows and the number of non-null positions are stored
- **Position bit-string:**
  - Bit-string for the column where non-null positions are set to 1 but null positions are set to 0
  - Accompanied by a list of the non-null values in the order of their appearance in the column
- **Position range:**
  - If there are *sequences of non-null values* in a column, the range of this sequence can be stored together with the list of values in the sequence
  - As metadata the total number of rows and the number of non-null positions are stored

> These three encodings *suppress nulls* and hence *reduce the size of the data set.* However, internally, *query evaluation* in the column store must be *adapted to the null suppression* technique applied

5.2 Column striping

The process of column striping decomposes a document into a set of columns:

- One column for each unique path in the document
- For each unique path, the values coming from different documents are written to the same column in order to be able to answer analytical and aggregation queries over all documents efficiently
- However we need some metadata to recombine an entire document as well as to be able to query it. These *metadata* are called:
  - *Repetition level* to handle repetitions of paths and it denotes at which level in the path the last repetition occurred. Moreover it can range from **0** to the **path length**:
    - **0:** no repetition has occurred so far
    - **Path length:** denotes that the entire path occurred for the previous field

- **Definition level** to handle non-existent paths of optional or repeated fields and it denotes the maximum level in the path for which a value exist. Only non-required fields are counted for the definition level

Chapter 6

Extensible Record Stores

Extensible record stores are database systems akin to **Google's BigTable system**.

- A "Bigtable" is a sparse, distributed, persistent multi-dimensional stored map
- It is indexed by **row key, column key and timestamp**
- A **value** in the map is an uninterpreted array of bytes, also called string
- MAP: (row:string, column:string, time:int64) ← string

These databases have tables as their basic data structure although with a highly flexible column management; they implement the concept of **column families** that act as containers for subsets of columns.

6.1 Logical Data Model

Extensible record stores do not use the normalization paradigm from RDBMSs. Instead they encourage a certain amount of **data duplication** for the sake of *better query locality* and more *efficient query execution*.

While the design in the relational model is centered around entities and later on normalization, an extensible record store first of all a typical query workload should be identified and data modeled around this workload accordingly.

What extensible record stores do to organize these key-value pairs is adding another structural dimension the **column family**:

- It groups columns that are often accessed simultaneously in a typical query
- The columns inside are called **column qualifier**
- The full **column name** consists of the column family name and the column qualifier
- **Value** for each column in a row
- **Row keys** link column of the same entity. However, there is no way of specifying foreign key constraints between different column families and no referential integrity is ensured

The **advantages** of column families are the following:

- They provide data locality by storing data inside a column family together on disk
- They provide dynamic addition of new columns (at run time)

And the following **characteristics**:

- They have to be created before used
- Columns families are fixed for a table
- There can be infinitely many columns in each column family

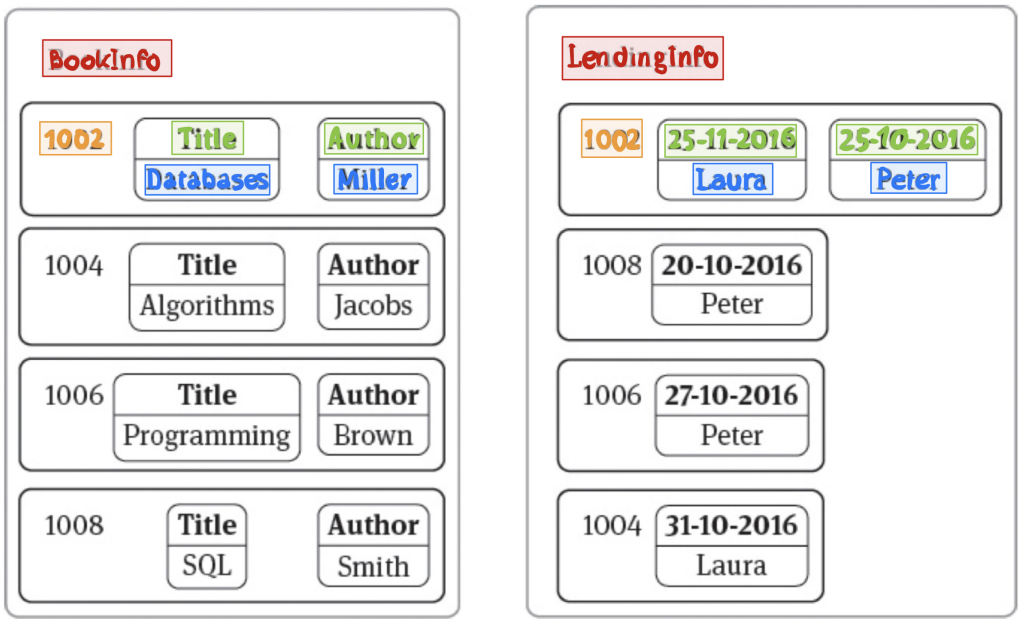


Figure 6.1: Column Families, Column Qualifiers, Values, Row Keys

At this point we also see the *flexibility* of how columns can be added to rows, indeed one row can have different columns than other rows inside the same column family. This is why extensible record stores are good at storing **sparse data:**

- An extensible record store just *ignores* values that are not present and *no null values* are included in rows.
- The **concatenation** of *row key*, *column family name* and *column qualifier* identifies a **cell** in the extensible record store
- The **full key** in order to access to a specific cell is in the form of:

$$rowkey : columnfamily : columnqualifier$$

Extensible record stores offer the convenient feature of **ordered storage:**

- While the relational data model is set-based and the order of the output basically depends on the DBMS
- Extensible record stores sort the data internally

- Inside a *column family*, the rows are sorted by their **row keys**
- Inside a *row* the columns are sorted by their **qualifiers**

Other extensible record stores (like Cassandra) offer data types for row keys and column qualifiers and hence sorting can be done according to the data type and may differ from the binary order.

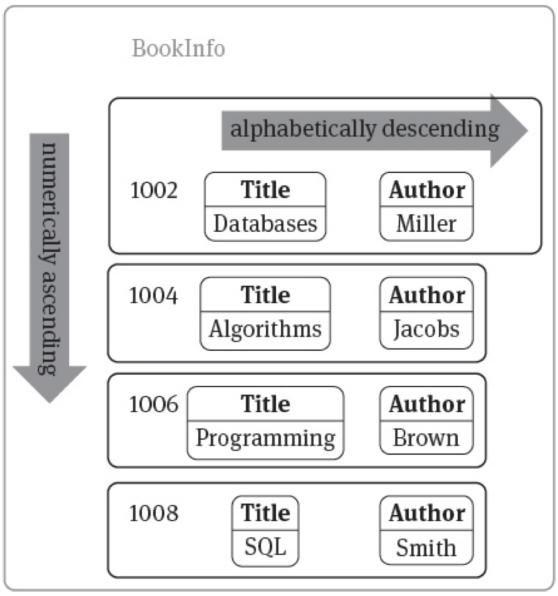


Figure 6.2: Ordering features of BookInfo

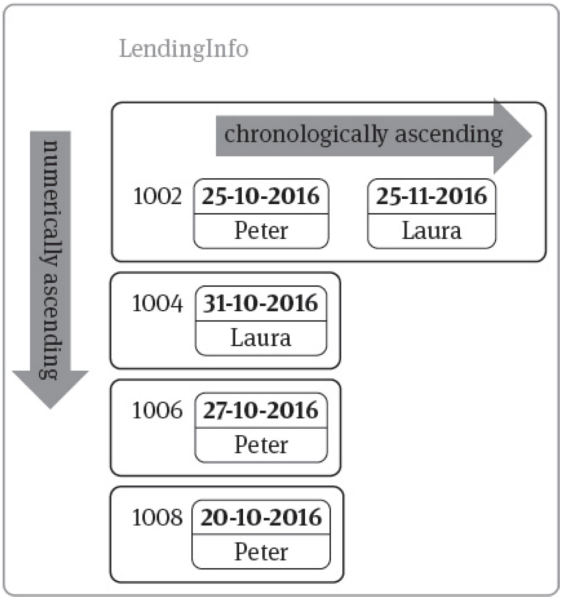


Figure 6.3: Ordering features of LendingInfo

With these ordering features, extensible record stores are particularly well-suited for identifying **contiguous sequences of columns**.

"What are the names of those readers who have to return books on or before 31-10-2016?"

To answer this query we have to find out what is following. The columns with column qualifiers less than or equal to 31-10-2016. But **due to the ordering** the matching columns are stored in a *consecutive range* and that we do not have to search further once we have reached a column qualifier greater than 31-10-2016.

Last but not least, some extensible record stores add *one more dimension* to the row-column family-column space: **time**.

- Each insert or update is accompanied by a **timestamp** and it can be specified by the user
- Extensible record stores provide an **automatic versioning** of column values
- When a read command *without an explicit timestamp* is issued on a cell, the *most recent* version is returned
- Versioning can further be configured by specifying a *maximum threshold* for the amount of stored version.
  - The **oldest version** will *automatically be discarded* once a new version is stored and the maximum value is exceeded
- Another option for **automatic discarding** is to specify a **time-to-live value** for each cell:
  - When the specified time span has *elapsed*, the corresponding *version* of the cell is **deleted**

Furthermore, extensible record stores usually do not make a distinction between inserts and updates. Instead, a **put command** is provided that checks whether there is an existing cell for the given key:

- If there is, the *value* for the key will be **updated**
- Otherwise a *new cell* is **inserted**

This is why this operation is sometimes called an **upsert**.

6.2 Physical storage

Extensible record stores use several techniques for **efficient query answering and recovery.**

6.2.1 Memtables and immutable sorted data files

Writing to memory tables and data files

Extensible record stores implement a *write-optimized storage model*

- All data records written to the on-disk store will only be appended to the existing records
- Once written, these records are read-only and cannot be modified, they are **immutable**
- Any modification must be simulated by appending a new record in the store

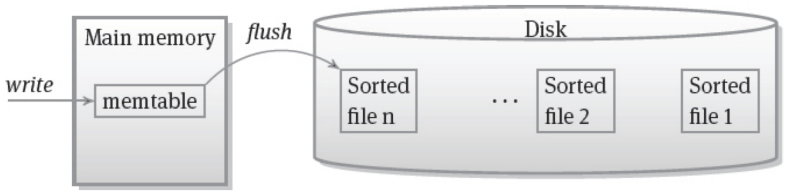


Figure 6.4: Writing to memory tables and data files

Reading to memory tables and data files

There are two types of read requests:

- *Get* also called **point query**. It accesses a particular row (identified by its row key).
- *Scan* also called **range query**. It iterates over a contiguous range of rows depending on some condition on the row key.

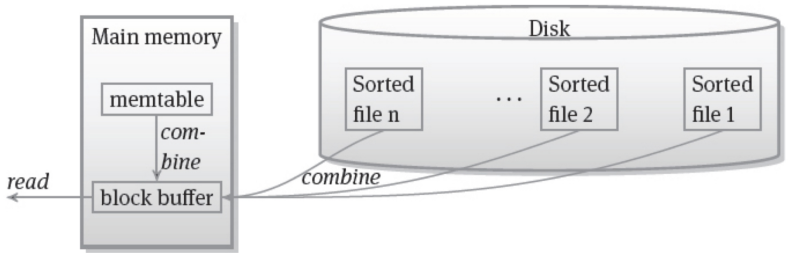


Figure 6.5: Reading to memory tables and data files

Consideration of both Reading and Writing operations

More specifically, *writing to* and *reading from* an extensible record store comprises the following steps:

- **Memtable:**
  - The most recent writes are collected in a main memory table of fixed size
  - Usually there is one memtable per column family
  - A record in the memtable is identified by its key (row key, column family, column qualifier and timestamp in milliseconds)
  - Timestamp can be chosen by the user
  - If no particular timestamp is specified in the put request, the current system time is used by default
- **Tombstones:**
  - Deletions are treated by writing a new record for a key
  - This record has no value assigned to it; instead a *delete marker* (called **tombstone**) is attached to the record
  - The tombstone masks all previous versions which will then be *invisible to the user*
  - A tombstone can for example **mark as deleted** either a *single version of a column* or an *entire column family*
- **Sorted data files:**
  - Once the memtable is filled, it is written to disk (**flushed**) and an entirely new memtable is started. With this action its records are sorted by key
  - The flushed sorted data files on disk are **immutable**
  - The **advantage of immutable data files** is that *buffer management* is a lot *easier*: no dirty pages
- **Combine upon read:**
  - The **downside** of immutable data files is that they **complicate the read process**
  - Retrieving all the relevant data that match a user query requires combining records from several on-disk data files and the memtable

Once the row keys to be accessed are identified, the result can be restricted to a subset of the columns inside each row. In some extensible record stores a set of versions of a cell can also be accessed. Usually, a *range of timestamps* can also be specified in a per-query basis;

Combining records from various on-disk data files and the memtable and identifying the most recent version of a column is *not trivial*, there may be a clash of timestamps.

- If the user specifies a key that already exists in a data file, a new record with the same key is appended to the memtable and later on flushed to a new data file
- When reading this key, several values for exactly the same key could be returned from different data files
- Desirable to determine a unique most recent value for each key
- **Unique sequence number** of the stored data files:
  - The record for a key that is contained in the data file with the highest sequence number is the most recently written one
- Additional difficulty comes since the extensible record store has to *interpret* the **time-to-live (TTL)** values of records as well as **tombstones** when retrieving and combining data from multiple sorted data files

Some *extra information* can be maintained for each data file to speed up the combine process; for example, the **range of row keys** in the file or the **minimum and maximum timestamp** can be stored as metadata of the file.

6.2.2 File format

Extensible records stores store the data in the on-disk data files in a certain format with the following properties:

- **Data blocks:**
  - An on-disk data file is composed of several data blocks
  - In some extensible record stores, a different block size can be used in each column family and hence the block size can be specified by the user when creating the column family
  - A block may also span multiple conventional memory pages (fixed size by the memory buffer and the OS)
  - Memory management is even more flexible in extensible record stores
- **Key-value pairs:**
  - A data block may contain one or more key-value pairs
  - Each key-value pair contains the entire key
  - This format hence is the foundation for the flexibility of extensible record stores because no fixed database schema is needed to interpret the data
  - This flexibility however comes at the price of *repetitious occurrences* of portions of keys
    * Row consists of several columns
    * Row key is contained in every record for each of these columns
    * Column families and column qualifiers are usually parts of keys in several different records

  - The **key** is followed by the **type** information, it determines if the record is a put (*upsert*) in which case a new value is appended or if it is a *deletion*

- **Index:**
  - Data files usually contain records for several keys and may become quite large
  - *Reading* records in a data file sequentially is a very inefficient method when searching for a single record for a given key in such a large data file
  - To **speed up** the process an index structure is maintained at the end of each file
  - The first row key on each block is inserted in the index
  - When searching for a given key in a data file, first of all the entire index of the data file is loaded into main memory
  - As not all row keys are maintained in the index, the index has to return either the entry for the exact search, or the index entry for the largest row key preceding the search key
  - In the later case the exact search key is not found in the index and we hence cannot be sure whether a record for the search key is contained in the data file or not

- **Trailer:** as the last component of a data file, a trailer contains management information, like where the index starts

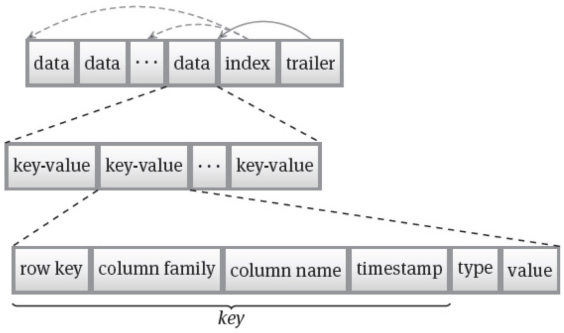


Figure 6.6: File format of data files

- **Multi-level index:**
  - Even though indexing is done at the block level and hence not all row keys are maintained in the index, the single index at the end of each data file might become quite large
  - Might slow down the read process because the entire index has to be loaded into main memory before accessing any key-value pairs
  - A multilevel index **splits the single index** into several *sub-indexes*:
    * One sub-index (*leaf index*) is stored at the end of each block
    * Only a small super-index (*root index*) pointing to the sub-indexes is stored at the end of the data file

6.3 Redo Logging

- The memtable is kept in volatile memory until it is eventually flushed to disk
- Data are only durable when stored in the on-disk data files and hence data that are contained in the memtable are vulnerable to failures of the database server
- **For example:** a crash of the server may wipe out the entire memory, or write errors may occur when flushing the memtable to disk
- When restarting the server the data in the memtable have to be recovered

Recovery is achieved with the help of an on-disk log file that keeps track of all records that are appended to the memtable but have not yet been flushed to the disk. This means that all data have be *written twice* (log file and memtable).

- Inside the log file, each record received a log sequence number (**LSN**) that maintains the *chronological order* of write operations
- Since the data is stored to the *on-disk log file* before appending them to the memtable, this process is called **write-ahead logging**

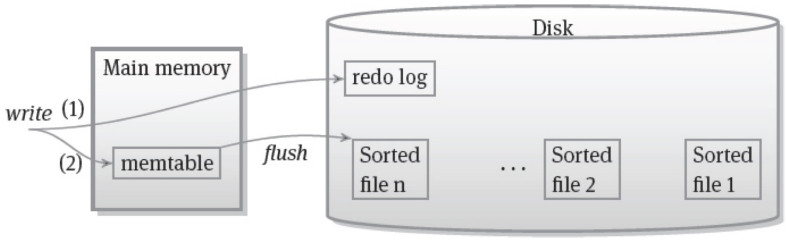


Figure 6.7: Write ahead log on disk

One peculiarity of extensible record stores is that they assume **redo-logging** to be *sufficient*:

- When restarting the database server after a failure the memtable is reconstructed from the log file by re-executing all the operations in the log as ordered by their LSNs.
- Remarking the fact that extensible record stores do not support transactions and long-lived transaction

Although the logging mechanism **adds an additional overhead** to the write process, it in fact **improves overall write performance**: while appending a record to the log file in *chronological order* is fast, sorting the records by key is *slower* and can be deferred until flushing the memtable.

Last but not least, writes to the same key can be melted (fuso) and only the most recent record has to be flushed to disk. In particular, **no record** for a key must be **written** to disk at all if an *upsert* for the key is *masked* by a **tombstone** for the key in the memtable.

6.3.1 Compaction

After some time, several flushes of memtables will have occurred and hence there will be quite a lot of data files stored on disk, in which probably contain some outdated records:

- Records whose time-to-live value has passed
- Records for which more recent versions exist
- Records for which the maximum numbe r of stored versions is exceeded
- Records which are masked by a tombstone

All this kind of data *slow down read* processes because they have to be loaded and compared with other records in the combine process of data retrieval. This is why a process called **compaction** was devised to remove any unwanted records and merge a set of data files into a new one.

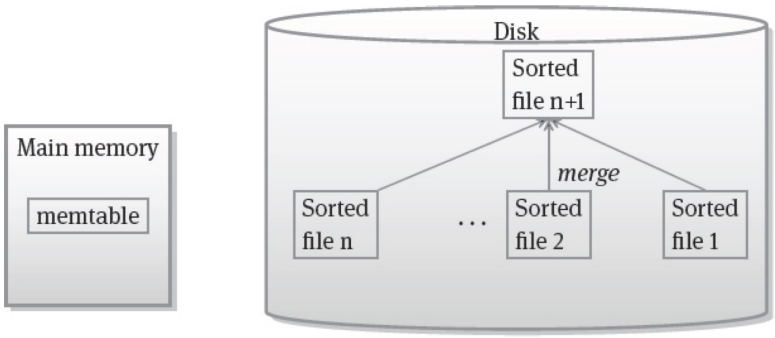


Figure 6.8: Compaction on disk

In this example image a set of data file is chosen for compaction, so their records are merged and the result is written to a new larger data file.

- **Minor compaction:** merges only a small subset of all data files
- **Major compaction:** merges all data files into a single new one

Several things have to be considered during compaction:

- Records have to be sorted by key thus the index have to be reconstructed
- Time-to-live values have to be interpreted to ignore invalid records
- If a tombstone is in a data file, all file marked by it and those written before the arrival of the tombstone can be ignored
  - Records that are masked by the tombstone but have been *written after* the insertion of the tombstone are *handled differently*: they are merged into the new data file but they still be masked by the tombstone if it is a *minor compaction*
  - **Tombstones** themselves can only be *deleted* during **major compaction**

  - So only after a major compaction more recent records for a key will be visible because they would previously be masked by the tombstone

- In some extensible record stores, *versioning settings* are also enforced during compaction: only a specified amount of versions for each key is kept at the maximum

- **Changing column family** settings can be done during *major compaction*

Compaction is demanding with respect to disk space. While the compaction process is running twice the disk space as occupied by the smaller data files. However they will be discarded after compaction effectively releasing the disk space.

Several **heuristics** can be applied when choosing data files **for minor compaction:**

- **Oldest files first:** chronologically ordered data file in which the oldest files with the lowest sequence number are chosen for compaction

- **Small files first:** to obtain a *homogeneous set* of data files, several similar sized smaller files are merged into one larger file

- **Compaction threshold:** the user can configure a minimum number of files for which a compaction is run

Note that with these heuristics, the same record may be chosen for minor compaction several times and the records unnecessarily often migrates from smaller files into larger files.

Furthermore, the **size of the resulting compacted** files *cannot be controlled*. To avoid this, **leveled compaction** has been developed:

- Data files are organized into levels

- Data files in lower levels are smaller than data files in higher levels

- A **flush** always moves the memtable to a data file in the *lowest level L0*

- *Subsequent compaction* steps move a record only from one level to the next, so amount of merges for each record is bounded by the number of levels

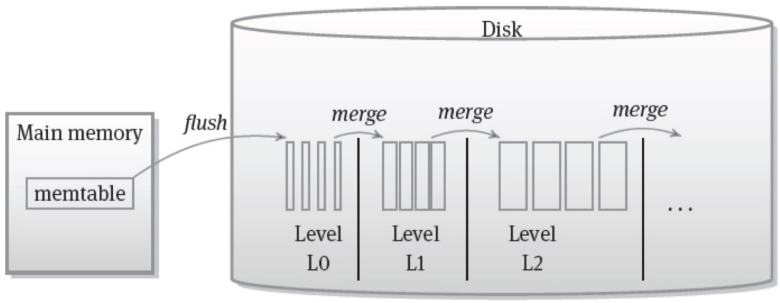


Figure 6.9: Level compaction

6.3.2 Bloom filters

A Bloom filter is a probabilistic method to determine set membership for a given value.

> For a given value c and a set S of values, the **Bloom filter** is a **small bit vector** with which we can decide whether c is *included* in S without actually searching for c in S. Of course it comes with a small probability of error.

- **True positive:** Bloom filter **correctly** reports a **match** and confirms that c is an element of S. So $c \in S$ holds
- **False positive:** Bloom filter **wrongly** reports a **match**, so it though that$c \in S$ but actually $c \notin S$
- **True negative:** Bloom filter **correctly** reports a **miss**, so $c \notin S$ holds;
- **False negative:** Bloom filter **erroneously** reports a **miss**, so it though that$c \notin S$ but actually $c \in S$

The only kind of error that could arises are **False positives**:

- When the Bloom filter reports a match we cannot be sure if this is true
- We have to actually search for c in S to verify whether $c \in S$ holds
- Or whether the Bloom filter wrongly believed c to be an element of S although in fact $c \notin S$ holds

For extensible record stores, Bloom filters *can be maintained* for all the row keys in a data file with the following positive effect: When searching for a given query key in the file, first the Bloom filter is accessed with the query key. If the Bloom filter reports a

- **Miss:** *true negative*, so we do not have to access any data
- **Match:** we have to load the index and search it for the query key to check whether the match was a *true positive* or a *false positive* (which means the record is not found in the data file)

Moreover for

- **Small data files:** a single Bloom filter can be appended at the end of the data file
- **Large data file** with lots of keys: a single Bloom filter will be too large. So this will result in performance delays when searching for a row key in the data file.

Bloom filter can be broken into pieces, a small Bloom filter "chunk" is maintained for the keys in each block; the Bloom filter chunk is then queried for the existence of a key before actually accessing data in the corresponding block.

A **Bloom filter** is a bit vector of a chosen length m with every position initialized to 0. The bit vector is accompanied by k different hash functions $(h_1, ..., h_k)$ where each hash function is assumed to map an arbitrary value c randomly to a number between 1 and m – that is, $(h_i(c) \in 1, ..., m)$.

For each number d between 1 and m, the probability that an input value c is mapped to d should be equal to $\frac{1}{m}$ this ca be written as $Prob(h_i(c) = d) = \frac{1}{m}$

The case that **two different input values** c and c' are mapped to the same value, so $h_i(c) = h_i(c')$ we will have a so called **collision**. Collisions are the reason why false positives can occur with Bloom filter.

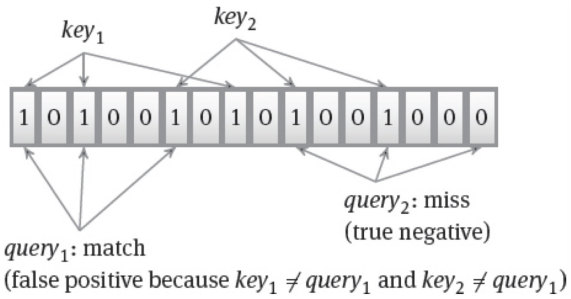


Figure 6.10: Bloom filter

The rate of false positive is approxiamted as:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right)^k \approx (1 - e^{\frac{-k \cdot n}{m}})^k$$

With k as the number of hash functions, m as the Bloom Filter length and n as the number of input keys.

Chapter 7

Graph Databases

A graph is a structure that not only can represent data but also connections between them; in particular, links between data items are explicitly represented in graphs.

7.1 Graphs and Graph Structures

Graphs structure data into a set of data objects and a set of links between these objects. The *data objects* are the **nodes** (vertices) and *links* are the **edges** (arcs).

> Graphs can store information in the nodes as well as on the edges.

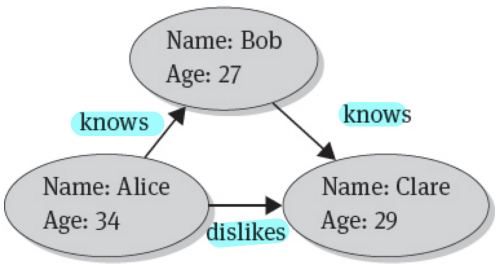
In a social network the **nodes** of a graph can store *information on people* and **edges** can store their *acquaintance* or express *sympathy* or *antipathy*.

Or in a geographic information systems **nodes** store information on *geographical locations* like cities and **edges** store for example the *distances* between the locations

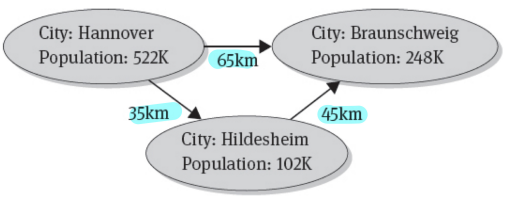
7.1.1 A Glimpse on Graph Theory

Mathematically speaking a graph $G = (V, E)$ consists on a set of nodes V and a set of edges E.

- The edge set E is a set of pairs of nodes and represents vertices that are connected by the edge
- The edge can be *directed* or *undirected*:



(a) A social network as a graph

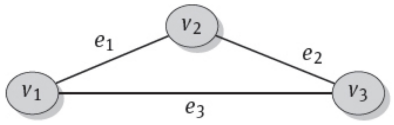


(b) Geographical data as a graph

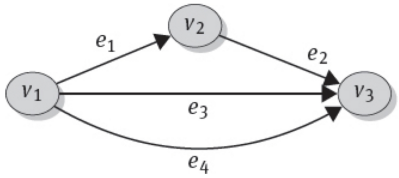
  - In a *directed* edge the pair of nodes is ordered where the first node is the **source** node of the edge and the last node is the **target** node
  - In the *undirected* case, order of the pair of nodes does not matter

Moreover, a graph is called *multigraph* if it has a pair of nodes that is connected by more than one edge. Now we have a look on a short summary on the graph types:

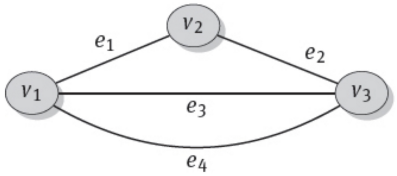
- **Simple undirected graph:**



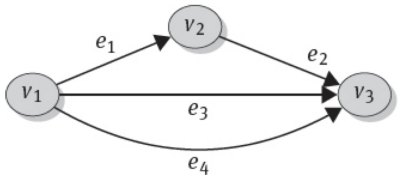
- **Simple directed graph:**



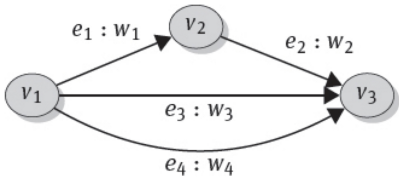
- **Undirected Multigraph:**



- **Directed Multigraph:**



- **Weighted graphs:**



7.1.2 Graph Traversal and Graph Problems

- A connection between to nodes consisting of intermediary nodes and the edges between them is called a **path**
- A path where starting node and end node are the same is called a **cycle**
- A **traversal** is a sort of navigation from a starting node to a specific destination nodes towards adjacent nodes. It could be:
  - *full* visiting each and every node in a graph
  - *partial* in which navigation may be restricted by a certain depth of paths to be followed

When traversing a graph, restrictions may be applied that have come to known as **graph problems**:

- **Eulerian Path:** a path that visits each edge exactly once;
- **Eulerian Cycle:** a cycle that visits each edge exactly once;
- **Hamiltonian Path:** a path that visits each vertex exactly once
- **Hamiltonian Cycle:** a cycle that visits each vertex exactly once
- **Spanning Tree:** a subset of the edge set V that forms a tree and visits each node of the tree

7.2 Graph Data Structures

When computing with graphs, the information on nodes and their connecting edges has to be represented and stored in an appropriate data structure. Two important terms in this field are **adjacency** and **incidence**

Two nodes are **adjacent** if they are neighbors (that is, there is an edge between them)

An edge is **incident** to a node, if it is connected to the node. Moreover if the edge is *directed*:

- It is **positively incident** to its *source* node
- And **negatively incident** to its *target* node

A node is incident to an edge, if it is connected to the edge.

7.2.1 Edge List

The graph can be stored according to the mathematical definition as a set of nodes V and a set (or list) of edges E. Edges could be represented as:

- *A set of sets* for **undirected graphs**
- *A multiset set* for **multigraphs**

We can compute the classical operation:

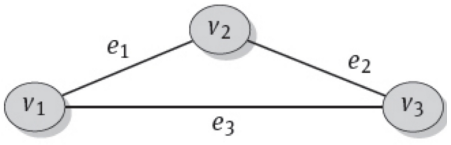
- **Addition** of an edge or nodes in which we simply add the element in the respective set
- **Deletion** by simply deleting the removed node or edge from the respective set
- **Search** the edge set performs well when one wants to retrieve all edges of the graph at once

However, the edge list representation is ineffcient in most use cases, like: looking for one particular edge or getting all neighbors of a given node requires *iterating over the entire edge list*

7.2.2 Adjacency Matrix

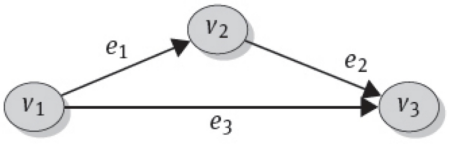
- For cardinality $|V| = n$ the adjacent matrix is $n \times n$ matrix
- Rows and columns denoted by $v_1, ..., v_n$
- **Simple undirected graphs**
  - If between v_i and v_j there is an edge we put a 1 in the corresponding cells (v_i, v_j) and (v_j, v_i) since *symmetry*
  - In case of *loops* v_i, v_i we write 2 in the diagonal cell v_i, v_i

	v_1	v_2	v_3
v_1	0	1	1
v_2	1	0	1
v_3	1	1	0



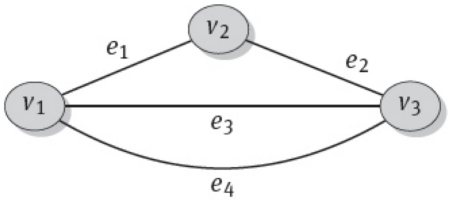
- **Simple directed graph**
  - If between v_i and v_j there is an edge we put a 1 in the corresponding cell (v_i, v_j) since *asymmetry*
  - In case of *loops* v_i, v_i we write 1 in the diagonal cell v_i, v_i

	v_1	v_2	v_3
v_1	0	0	0
v_2	1	0	0
v_3	1	1	0



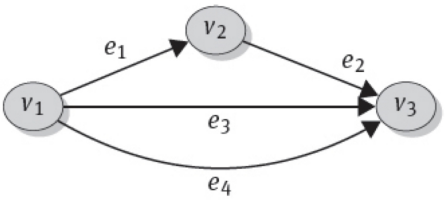
- **Undirected multigraph**
  - If there are k edges between v_i and v_j we put a k in the corresponding cells (v_i, v_j) and (v_j, v_i) since *symmetry*
  - In case of k *loops* v_i, v_i we write $2 \cdot k$ in the diagonal cell v_i, v_i

	v_1	v_2	v_3
v_1	0	1	2
v_2	1	0	1
v_3	2	1	0



- **Directed multigraph**
  - If there are k edges between v_i and v_j we put a k in the corresponding cell (v_i, v_j) since *asymmetry*
  - In case of k *loops* v_i, v_i we write k in the diagonal cell v_i, v_i

	v_1	v_2	v_3
v_1	0	0	0
v_2	1	0	0
v_3	2	1	0



Advantages

- Quick lookup of existence of a single edge by looking at the bit value of the matrix
- Quick insertion of new edge by incrementing the bit in the matrix cell

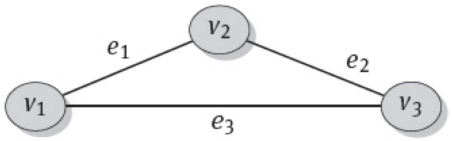
Disadvantages

- Adding a new node requires insertion of a new row and a new column and finding all neighbors results in a scan of the entire column
- $n \times n$ matrices are heavy
- Unnecessary information, indeed lots of 0s in the rows
- No hyperedges can be stored (edges that connects a set of nodes)

7.2.3 Incidence Matrix

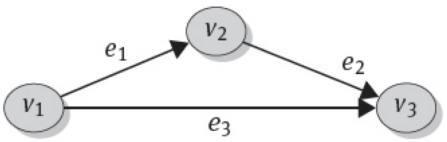
- For cardinality $|V| = n$ and $|E| = m$ the incidence matrix is $n \times m$ matrix
- Rows denote vertices $v_1, ..., v_n$ and columns denote edges $e_1, ..., e_m$
- **Simple undirected graph**
  - If vertices v_i is connected to edge e_j we write 1 in the corresponding cell (v_i, e_j)
  - In case of *loop* $e_j = \{v_i, v_i\}$ we write 2 in the corresponding cell (v_i, e_j)

	e_1	e_2	e_3	
v_1	1	0	1	2
v_2	1	1	0	2
v_3	0	1	1	2
	2	2	2	



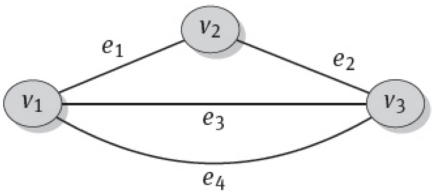
- **Simple directed graph**
  - For edge $e_k = (v_i, v_j)$ we write -1 in cell (v_i, e_k) and 1 in cell (v_j, e_k)
  - In case of *loop* $e_k = \{v_i, v_i\}$ we write 2 in the corresponding cell (v_i, e_k)

	e_1	e_2	e_3
v_1	-1	0	-1
v_2	1	-1	0
v_3	0	1	1



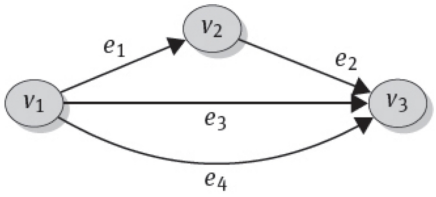
- **Undirected multigraph**
  - If vertices v_i is connected to edge e_j we write 1 in the corresponding cell (v_i, e_j)
  - In case of *loop* $e_j = \{v_i, v_i\}$ we write 2 in the corresponding cell (v_i, e_j)

	e_1	e_2	e_3	e_4
v_1	1	0	1	1
v_2	1	1	0	0
v_3	0	1	1	1



- **Directed multigraph**
  - For edge $e_k = (v_i, v_j)$ we write -1 in cell (v_i, e_k) and 1 in cell (v_j, e_k)
  - In case of *loop* $e_k = \{v_i, v_i\}$ we write 2 in the corresponding cell (v_i, e_k)

	e_1	e_2	e_3	e_4
v_1	-1	0	-1	-1
v_2	1	-1	0	0
v_3	0	1	1	1



Advantages

- Only existing edges are stored, there is no column with only 0 entries
- Hyperedges can be stored

Disadvantages

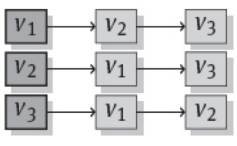
- Insertions of new vertices and edges are costly, since the addition of a row or a column
- Determining all neighbors for one vertex requires scanning the entire row and for each non-zero entry
- $n \times n$ matrices are heavy
- Unnecessary information, indeed lots of 0s in the columns

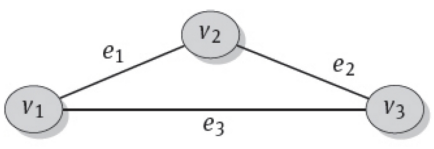
Note that **checking the existence of an edge** is more involved for the *incidence matrix* than for the adjacency matrix: we have to check whether there is a column with appropriate non-zero entries for the source node's row and the target node's row.

7.2.4 Adjacent List

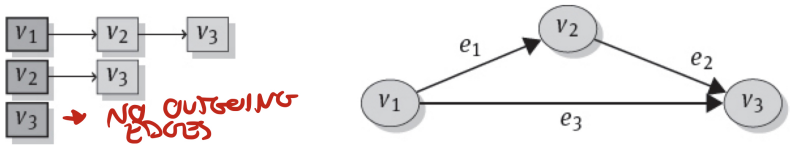
It stores the vertex set V and for each vertex one stores a **linked list of neighboring**

- **Simple undirected graph:** each edge is stored in the adjacency list of both its vertices

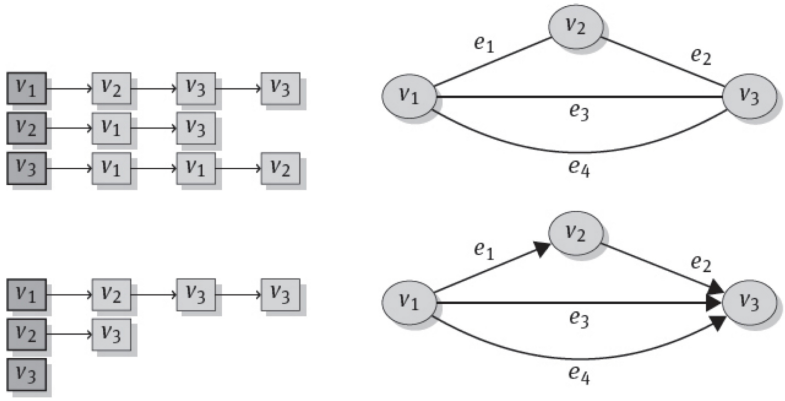




- **Simple directed graph:** it stores only outgoing edges



- **Multigraphs:** in both the directed as well as the undirected case, nodes can occur multiple times in an adjacency list



Advantages

- Quick insertion of new vertices and edges
- Quick lookup of all neighboring vertices
- No storage overhead occurs
- Hyperedges can be stored

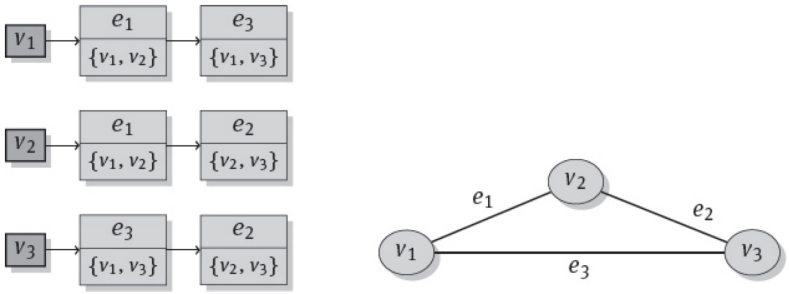
Disadvantages

- Checking existence of a single edge requires a full scan of the adjacency list of the source node

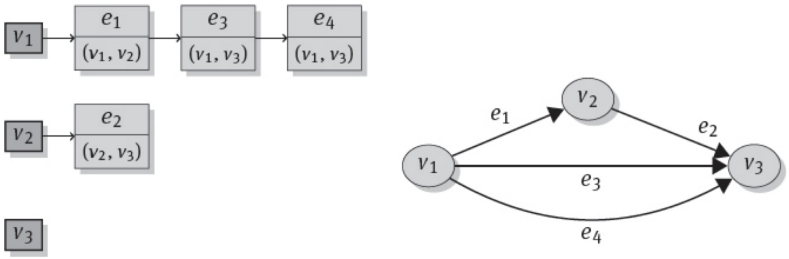
7.2.5 Incidence List

With an incidence list, you store the **vertex set** V and for each vertex you store a **linked list of incident edges**

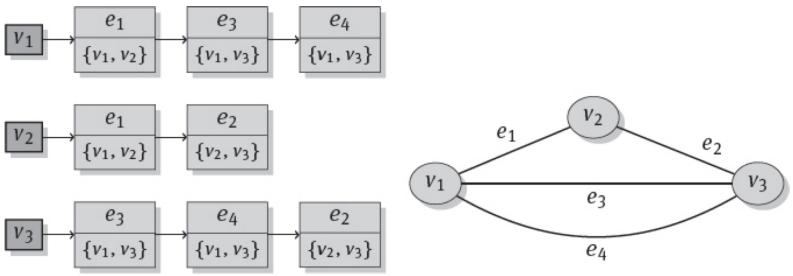
- When the edge is *directed* the edge object contains information on its **source** node and its **target node**
- When the edge is *undirected* no difference is made between source and target nodes
- **Simple undirected graph:** each undirected edge is contained in the incidence lists of its two connected nodes



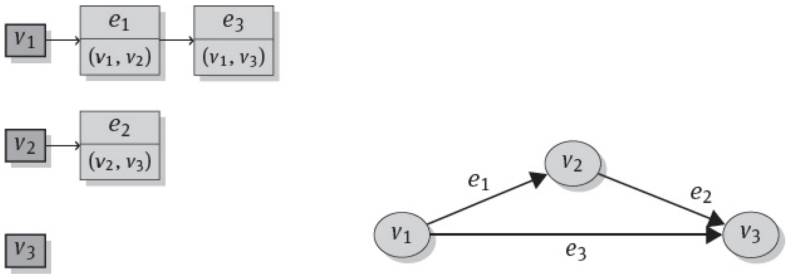
- **Simple directed graph:** it suffices to store only outgoing edges in the incidence list as long as only a forward traversal of the edges is needed. In this case, it is advantageous to store all incident edges in a node's incidence list to allow for both forward traversal of the outgoing edges and backward traversal of the incoming edges. We could have two incidence list¿
  - One for the **outgoing** edges for the forward traversal
  - One for the **incoming** edges for the backword traversal



- **Multigraphs:** in both directed as well as the undirected case, each edge has its own identity and hence is stored separately.
  - For the **undirected** case each edge contains pointer to the incident nodes



  - For the **directed** case each edge would have a pointer to its source node as well as one pointer to its target node



In practical implementations, the incidence list would be **stored inside a node object** as a **collection of pointers** to incident edge objects – potentially – in the directed case – one collection for incoming and one collection for outgoing edges to allow for both forward and backward traversal.

7.3 The Property Graph Model

- The basic storage structure usually is a directed multigraph
- We must be able to **store information** inside the *node* as well as along the *edges*
- We must be able to distinguish different kind of nodes and edges. Point achieved by the **multi-relational** graph where **types** are introduced for nodes and edges.
  - Each node is labeled with the **node label** that correspond to the node type
  - Each edge is labeled with the **edge label** that correspond to the edge type
- A type defines **attributes** for the corresponding nodes and edges. So an attribute definition must contain a name for the attribute and it mus specify a domain of values
- Like *name:value* pairs describe **proprieties** of a node or edge
- **Edge labels** between any two nodes should be **unique**
- Each node has a system-defined **unique identifier**

A propriety graph G can be defined as $G = (V, E, L_V, L_E, ID)$:

- V set of **nodes**
- E set of **edges**
- L_V set of **node labels** st to each label $l \in L_E$ we can assign a set of attribute definitions

- L_E set of **edge labels** st to each label $l \in L_E$ we can assign a set of attribute definitions
- ID set of identifiers that can uniquely be assigned to nodes and edges

A specific node $v \in V$ has the following definition: $v = (id, l, P)$:

- $id \in ID$
- $l \in L_V$
- P is a set of proprieties, st each **propriety** $p \in P$ is a *name:value*-pair st:
  - The *name* correspond to an attribute name defined by the node type
  - The *value* is a valid value taken from the teh attribute domain

Similarly an edge $e \in E$ is defined like so $e = (id, l', P, source, target)$:

- l' is an edge label from L_E
- The proprieties in P correspond to an attribute definitions of this edge type

Example

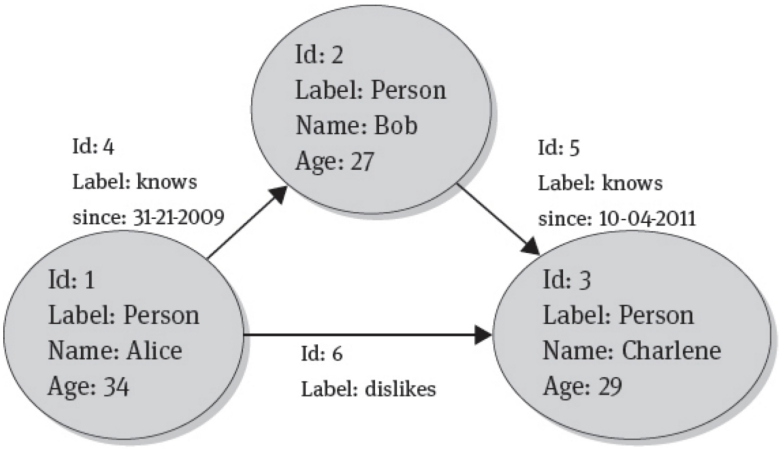


Figure 7.2: A property graph for a social network

- The node set is $V = \{v_1, v_2, v_3\}$
- The edge set is $E = \{e_1, e_2, e_3\}$
- The node labels are $L_V = \{Person\}$
- The node type definitions are $t_{Person} = \{Person, A_{Person}\}$ where the attribute definitions are $APerson = \{(Name,\ String),\ (Age,\ Integer)\}$
- The edge labels are $L_E = \{knows, dislikes\}$

- The edge type definitions are $t_{knows} = \{knows, A_{knows}, \{Person\}, \{Person\}\}$ and $t_{dislikes} = \{dislikes, \emptyset, \{Person\}, \{Person\}\}$, where $Aknows = \{(since, Date)\}$

The specification of nodes and edges are the following:

- $v_1 = \{1,\ Person,\ \{Name:\ Alice,\ Age:\ 34\}\}$
- $v_2 = \{2,\ Person,\ \{Name:\ Bob,\ Age:\ 27\}\}$
- $v_3 = \{3,\ Person,\ \{Name:\ Charlene,\ Age:\ 29\}\}$
- $e_1 = \{4,\ knows,\ \{since:\ 31\text{-}21\text{-}2009\},\ 1,\ 2\}$
- $e_2 = \{5,\ knows,\ \{since:\ 10\text{-}04\text{-}2011\},\ 2,\ 3\}$
- $e_3 = \{6,\ dislikes,\ ;,\ 1,\ 3\}$

In addition we specify an additional requirement: **there may never be two edges with the same label between two nodes.** Like the following example

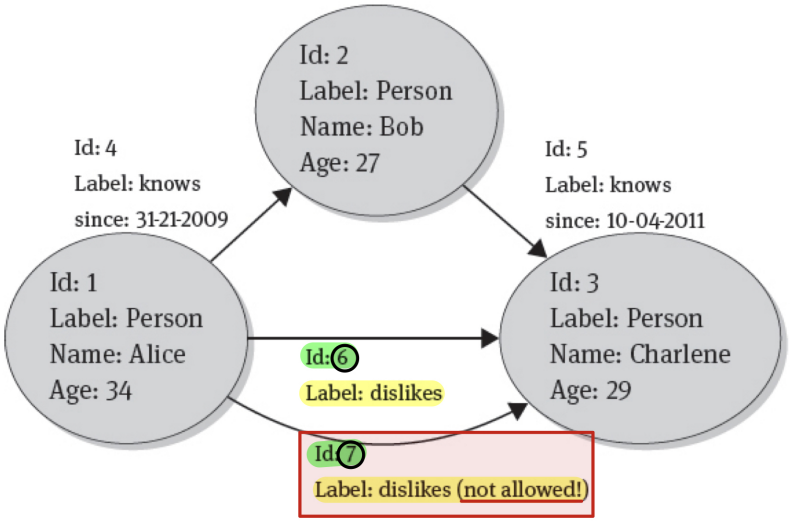


Figure 7.3: Violation of uniqueness of edge labels

Chapter 8

Distributed Databases

For several decades, centralized database management systems running on a single database server have been predominant, for these four main reasons:

- *Complexity* of a single-server system was lower and administration easier
- *Evaluate short queries* and *infrequent data modification*
- *Slow network* so sending large amount of data was too costly
- *Parallelization* required rewriting a query into subqueries and recombining the results, overhead diminished the positive effects of a parallel execution of subqueries

Whenever there were more demanding requirements:

- **Scaling up** or **Vertical scaling** equip the single database server with higher hardware capacity
  - Amount of data and queries a single database server can handle is limited
  - A single server is always a single point of failure
- **Scaling out** or **Horizontal scaling** connecting several cheaper servers in a network
  - The only option to improve the throughput and latency of a database system
  - Disadvantage of cost of coordination and synchronization of the database servers
  - However this last one pays off for large scale systems or global enterprises with several data centers

8.1 Scaling Horizontally

- The ability of a database system to *flexibly scale out by distributing data in a server network* is called **horizontal scalability**
- Servers can work independently

- Also called **shared-nothing architecture**
- The most common use case today is a distributed database on a shared-nothing architecture
  - A distributed database management system (DDBMS) that runs on a network of independent servers
  - The servers need not be large
  - They consist of cheaper commodity hardware so that each server can easily be replaced by a new one

DBMS are benefical also when aiming for improved availability and reliability in smaller scaled systems:

- **Load balancing:** user queries and other processes should be assigned to the servers in the network such that all servers have approximately the same load
- **Flexible scalability:** servers may flexibly leave and join the network at any time
- **Heterogeneous nodes:** servers may flexibly leave and join the network at any time
- **Symmetric configuration:** every node is configured identically to the others;
- **Decentralized control:** peer-to-peer algorithms for data management improve failure tolerance of a DDBMS

8.2 Distribution Transparency

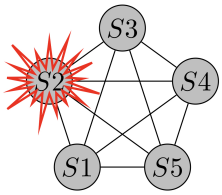
For a user it must basically be transparent how the DBMS internally handles data storage and query processing in a distributed manner.

- **Access transparency:** uniform query and management interface to users independent of the structure of the network
- **Location transparency:** the distribution of data in the database system is hidden from the user
- **Replication transparency:** if several copies of a data item are stored on different servers the user should not be aware of this
- **Fragmentation transparency:** if a large data set has to be split into several data items and the distributed database system does this splitting internally and the user can query the database as if it contained the entire unfragmented data set
- **Migration transparency:** if data items have to be moved from one server to another, this should not affect how a user accesses the data

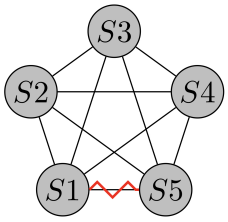
- **Concurrency transparency:** when multiple users access the database system, their operation must not interfere or lead to incorrect data in the database system
- **Failure transparency:** as a distributed database system is more complex than a centralized one

8.3 Failures in Distributed Systems

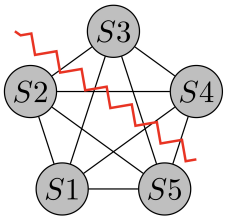
- **Server failure:** a DBS may fail to process messages due to notwork component crash or a self crash



- **Message failures:** message through the network links may be delayed or lost during times of high congestion
- **Link failure:** a communication link between two servers may be unable to transmit messages. Note that a *link failure* **can cause** a *message failure*



- **Network partition:** when the network is splitted into two or more subnetworks that are unable to communicate



Node Failure can be categorized into:

- **Crash failures:** is a permanent failure of a server and corresponds to aborting a communication protocol. Once the server crashed, it will never resume operation
- **Omission failures:** an omission failure corresponds to not taking some action

- **Commission failures:** corresponds to taking an action that is not correct according to a communication protocol

Moreover we note that:

- Crash failures are a special case of omission failures.
- The union of omission and commission failures is called **Byzantine failures**
- The term **non-Byzantine failures** usually refers to omission failures but in addition explicitly also covers duplication and reordering of messages

Distributed DBMSs have to provide a high level of **fault tolerance**, indeed, a distributed system may in general be design based on a certain **failure model** which describes the set of failures that the system can tolerate.

- **Fail-stop model:** all server failures are crash failures that permanently render the server unavailable
- **Fail-recover model:** a server may halt but it may later resume execution. The resumption could be:
  - In the state before it was halted
  - From scratch

8.4 Epidemic Protocols and Gossip Communication

Due to the many properties the propagation of information in the network is difficult to manage. In the *simplest scenario*

- Whenever new information is received by one server, the server sends a notification to all the other servers he know
- But the initiating server might not be aware of all servers currently in the network
- Some of his messages might be lost due to network failures

An alternative is when the DBS can be seen as participants in a *peer-to-peer network* where there is *no central coordinator*. **Epidemic protocols** are a category of peer-to-peer algorithms, where information is spread like an infection all over the network

An other application is **membership** of peers n the network:

- Each server has to maintain a list of names
- This list can be kept up-to-date by an epidemic algorithm
- Servers exchange their membership lists in a peer-to-peer fashion so that the information which servers are part of the network slowly spreads over the entire network

With an epidemic algorithm, servers in the network pass on a message **like an infection**. A **message** notification that some new server has joined the network, so the servers that receive this message update their membership list. We have three types of nodes:

- **Infected nodes:** are servers that have received a new message that they want to pass on to others
- **Susceptible nodes:** are nodes that so far have not received the new message
- **Removed nodes:** are nodes that already have received the message but are no longer willing to pass it

There are three different communication modes that can be applied in epidemic algorithms:

- **push-only:** an infected server contacts another server and passes on all the new messages it has received. Infected server → susceptible server
- **pull-only:** a susceptible server contacts another server and asks for new messages. Susceptible server → infected server
- **push-pull:** one server contacts another server and both exchange their new messages. Both servers have the same state

A synonym for epidemic message exchange is **gossiping:** the term expresses that messages spread in a server network like rumors in human communication.

Two variants of epidemic algorithms for database updates are:

- **Anti-entropy** is a periodic task that is scheduled for a fixed time span, for example every minute. It is called also *simple epidemic* since any server is either susceptible or infective and the infection process *does not degrade* over time or due to some probabilistic decision.
- **Rumor spreading** is an infection that can be triggered by the *arrival of a new message* or it can be run periodically. Here the infection have several rounds, in each one a server chooses a set of communication partners (called *fanout*). It is a *complex epidemic* because infection of other servers is a dynamic process: amount of server decreases every round.

This decrease of infections can be varied as follows:

- *Probabilistic:* after each exchange with another server, the server stops being infective with a certain probability
- *Counter-based:* After a certain number k of exchanges, the server stops being infective. Therer are two cases:
  - *Infect-and-die* the number k is equal to the fan-out
  - *Infect-forever* the number k is infinite and the server never stops
- *Blind:* the server becomes removed without taking the feedback of communication partners is into account
- *Feedback-based:* the server becomes removed if it notices that the communication partners already have received the new message

8.4.1 Hash Trees

A major issue with epidemic protocols is how two servers can identify those messages in which they differ. For example a complete comparison of all messages is not feasible as this would slow down the epidemic process tremendously.

A simple improvement is to use a **list of hash values**

- Comparison of hash values is faster
- The downside, the hash values have to be computed and still the list of hash values has to be compared sequentially

A step further could be take using a **hash tree**:

- It starts with a hash of each message in a leave node, and then iteratively concatenates hashes
- For *inner node*, closer a hash value is to the root, the more leave it covers
- The last hash value at the root of the tree is called the **top hash**

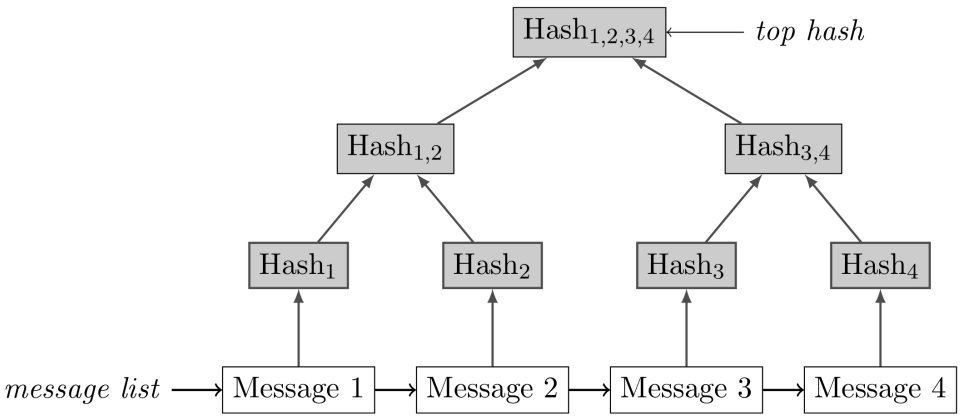


Figure 8.1: A hash tree for four messages

Now, with a hash tree, message list comparison is improved:

- When the two top hashes are identical, the two message lists are identical
- If the top hashes differ, we go the the next level of the tree and compare the hash values there identical
- Whenever we encounter an inner node that has identical hash values in the two hash trees under comparison, then we know that the messages below this inner node are identical
- As long as hash values differ for an inner node, we have to go one level deeper and compare the hash values of the child nodes
- When we reach a leaf node with different hash values, we have identified a message on which the two message lists differ

In order to avoid unnecessary hash comparisons, we have to ensure identical root hashes for identical message list. And could be done by:

- Let the two servers agree on a sorting order, sort all messages according to this order and then compute the hash tree just before the comparison
- Or make each server precompute hash trees for any possible sorting order of the messages. And for comparison the servers then just have to find those two trees with the same sorting order

Chapter 9

Data Fragmentation

In a distributed database system, two major questions are:

- How the entire set of data items in the database can be split into subsets → **data fragmentation (sharding / partitioning)**
- How the subsets can be distributed among the database servers in the network → **data allocation**

9.1 Properties and Types of Fragmentation

The query behavior of users plays an important role for the quality of fragmentation and allocation, like the:

- Type of access
- Access patterns
- Affinity of records
- Frequency of accesses
- Accesses duration

Once a good fragmentation and allocation have been established, a distributed database can take **advantage** of:

- Data locality
- Minimization of communication costs
- Improved efficiency of data management
- Load balancing

However on average a fragmented database system has to live with several **disadvantages**:

- Queries that involve subqueries over different fragments are costly because the global query has to be split into subqueries

- Distributed transactions are extremely difficult to manage

Several criteria are important when designing a distributed database with fragmented data sets. The most important is that a given data fragmentation must be correct, in the sense that the original data must be *entirely reconstructed*. When the data set is recombined from the fragments, the following two **correctness properties** are required:

- **Completeness:** none of the original data records is lost during fragmentation and hence no data is missing in the reconstructed data set
- **Soundness:** no additional data are introduced in the reconstructed data set

9.2 Data Allocation

For fragmented data a major problem is how to *distribute the fragments* over the database servers. A **data allocation mechanism** has to decide which fragment should be stored on which server, thus **load balancing** has to be taken into account. There are several forms of data allocation:

- **Range-based allocation** relies on range-based fragmentation. When a range-based fragmentation is obtained the identified ranges have to be assigned to the available servers
- **Hash-based allocation** uses a hash function on the input fragments to determine which server each fragment is assigned to. With hash-based allocation the distribution of the records among the servers is usually more balanced because the hash function distributes its input values well over its entire output hash values
- **Cost-based allocation** describes the data allocation task as an optimization problem

9.2.1 Consistent Hashing

Consistent hashing is an hash-based allocation schema which provides a better and more flexible distribution of the records among a set of servers.

- A hash function is computed on each input fragment
- The hash values are now seen as a ring that wraps around: when we have reached the highest hash value we start again from 0
- A hash value is computed not only for each fragment but also for each database server

By computing a hash value for a database server, **each server** has a *fixed position on the ring*; the advantage of these hash values is that they presumably distribute the servers uniformly on the ring.

A widely used allocation policy is then to store data on the next server on the ring when looking in clockwise direction.

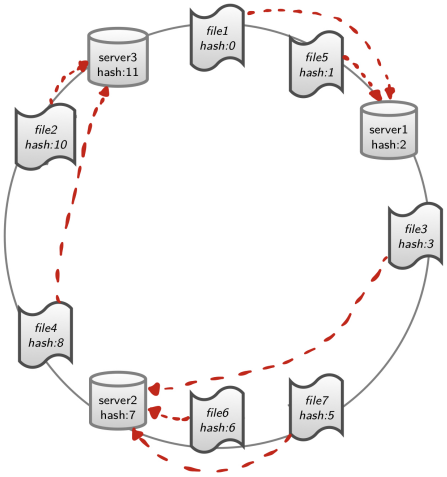


Figure 9.1: Data allocation with consistent hashing

A major advantage of consistent hashing is its *flexible support* of **additions** or **removals** of servers.

- Whenever a server **leaves** the ring: all the data that it stores have to be moved to the next server in clockwise direction

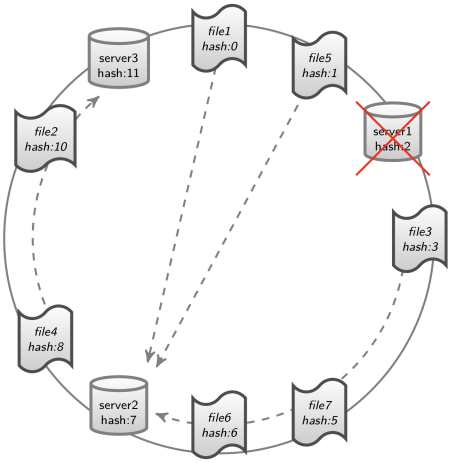


Figure 9.2: Server removal with consistent hashing

- Whenever a server **joins** the ring: the data with a hash value less than the hash value of the new server have to be moved to the new server

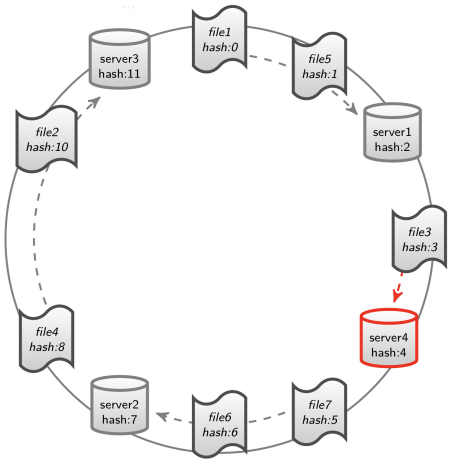


Figure 9.3: Server addition with consistent hashing

An important tool to make consistent hashing *more flexible* is to have not only one location on the ring for each physical server but instead to have *multiple locations*: these locations are then called **virtual servers**. Virtual servers improve consistent hashing in the following cases:

- The virtual servers (of each physical server) are spread along the ring in an arbitrary way. So, all servers have a better spread on the ring which leads to a better data distribution
- Heterogeneous servers are supported. So a server with less capacity can be represented by less virtual servers than a server with more capacity
- New servers can be gradually added to the ring: instead of shifting its entire data load onto a new server at once, virtual servers for the new server can be added one at a time. In this way the new server has tie to start up slowly and take in its full load step by step until its full capacity is reached.

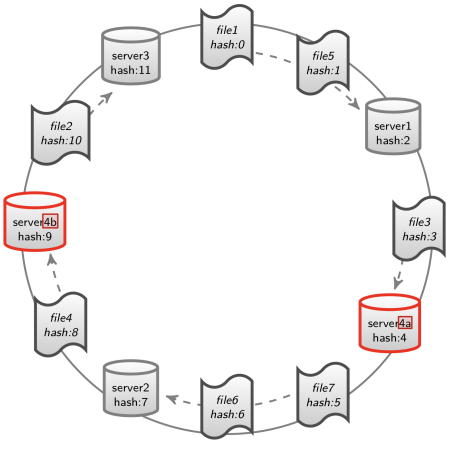


Figure 9.4: Virtual server with consistent hashing

Chapter 10

Replication And Synchronization

Replication refers to the concept of storing several copies of a data record at different database servers. These *copies* are called **replicas** and the *number of copies* is called the **replication factor**. When applying replication to large data sets:

- First a fragmentation of the data set is obtained
- Fragments are replicated among a distributed database system
- Replication improves reliability and availability of a distributed system because replicas can serve as backup copies whenever one of the servers fails and becomes unavailable
- It enables load balancing or data locality
- This kind of concurrent accesses to different replicas lead to consistency problems

10.1 Replication Models

Replication has several **advantages:**

- It improves the **reliability** by offering higher data availability
- It offers **lower latency** by enabling load balancing, data locality and parallelization

These two advantages implies respectively two a major **disadvantages**:

- **Consistency problems**: because when a user updates a data record at one server, network delays or even network failures may prevent the database system from updating all other replicas of the data record quickly
- **Concurrency problem:** where two or more users might concurrently update the same data record on different replicas

The two basic models of replication are **master-slave** and **multi-master** replication. While the *consistency problem* exists for both, the *concurrency problem* is **avoided** in the **master-slave** case but at the cost of higher latency for write requests

10.1.1 Master-Slave Replication

- In master-slave replication, *write* requests are handled only by a *single server* that is called the **master**.
- After a write, the master is responsible for *updating* all other servers that hold a replica, called **slaves**
- Read requests can be accepted by both the master and the slaves

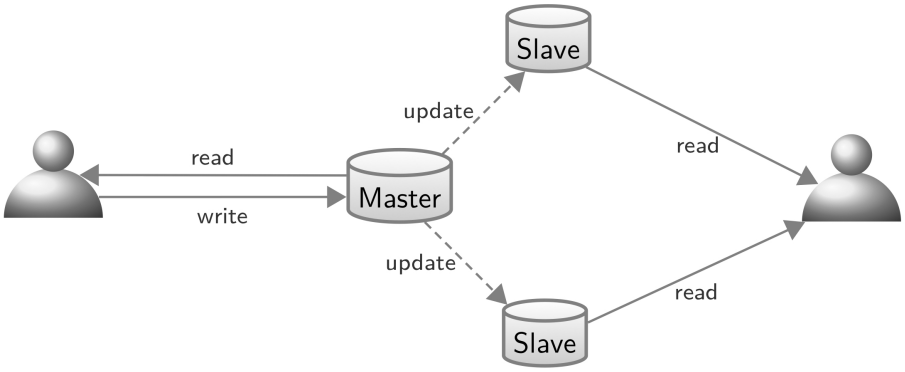


Figure 10.1: A Master-slave replication

Master-slave replication offers enough **redundancy** in case of a **master failure**: when the master fails, *one of the slaves* can be **elected** to be the **new master** and all write request are redirected to it.

Having a *single master server* for all write requests in the database system is a **bottleneck** that slows down the processing of writes tremendously.

- A solution is to *partition* the set of all data records into **disjoint subsets** and to each such *subset* assign *one server* as the **master server**
- In combination with data partitioning, data records in the same partition are copied to the same replication servers and one of the servers is designated master for the entire partition while the others act as slaves

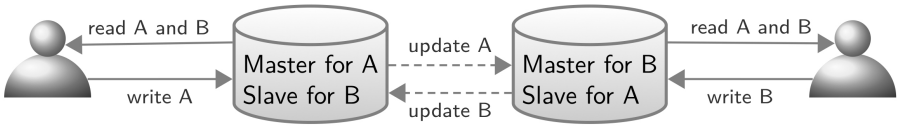


Figure 10.2: Master-slave replication with multiple records

10.1.2 Multi-Master Replication

- When all servers holding a replica of a data record can process write request, they all act as masters for the data record, this is the case of **multi-master replication** or **peer-to-peer replication** (based on the fact that the masters are peers with identical capabilities and they have to synchronize one with the other)

- Higher write availability than master-slave replication, because clients can contact any replica server with a write request, parallel requests
- All servers accept write and read requests for a data item
- The servers have to regularly synchronize their state among themselves
- Due to the **consistency problem:**
  - Clients may retrieve *outdated data* whenever the replica answering the client's read request has *not finished the synchronization process*
- Due to the **concurrency problem:**
  - Replicas may be in conflict when different clients wrote to different replicas without a synchronization

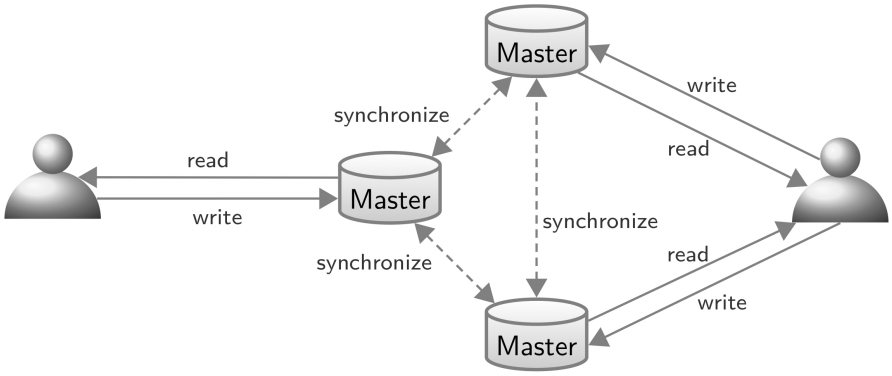


Figure 10.3: Multi-master replication

Chapter 11

Polyglot DB Arch

When designing the data management layer for an application, several of the identified database requirements may be contradictory. Regarding the data model, some data might be of a different structure than other data.

11.1 Polyglot Persistence

Instead of choosing *just one single DBMS* to store the entire data, so-called **polyglot persistence** could be a viable option to satisfy all requirements towards a modern data management infrastructure.

- It denotes that one can choose as many databases as needed so that all requirements are satisfied
- Optimal solution when backward-compatibility with a **legacy application** must be ensured

Polyglot persistence however comes with severe **disadvantages:**

- There is no unique query interface or query language, thus there is not a unique database access method
- Cross-database consistency is a major challenge, and in case data are duplicated the duplicates have to be updated or deleted in unison

It should obviously be avoided to push the burden of all of these query handling and database synchronization task to the application level. Thus the *integration layer* is introduces to take care of processing queries.

- Decomposing queries in to several subqueries
- Redirecting queries to the appropriate databases
- Recombining the results obtained from the accessed databases

Finally the *integration layer* should ensure **cross-database consistency**: it must synchronize data in the different databases by propagating additions, modifications or deletions among them.

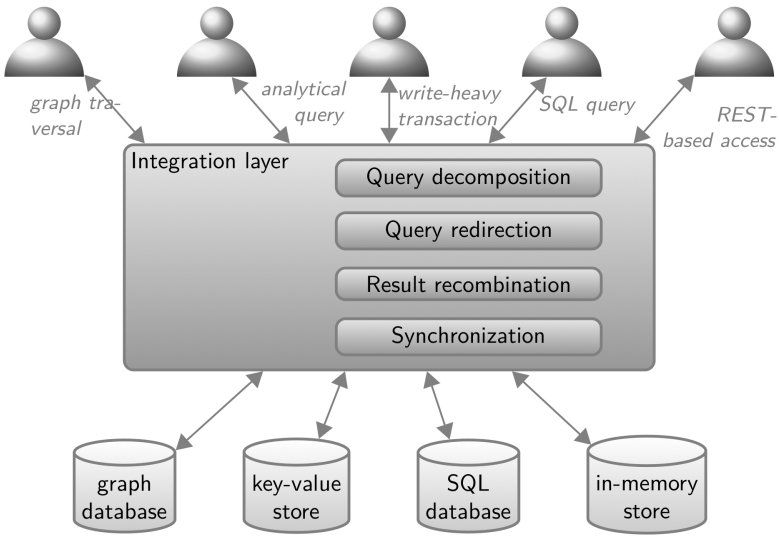


Figure 11.1: Polyglot persistence with integration layer

11.2 Lambda Architecture

When real-time (stream) data processing is a requirement, the combination of the following two features might be appropiate:

- A slower batch processing layer
- A quiker stream processing layer

This architecture has been recently termed **lambda architecture**, it major feature is that it processes a continuous flow of data in the following three layers:

- **Speed Layer** collects only the most recent data and as soon as data ave been included in the other two layers, it can be discarded from the speed layer. Moreover it computes results over its dataset and delivers teh results in several **real-time views**
- **Batch Layer** stores all data in an append-only and immutable way in a so-called *master dataset*. It evaluates functions over the entire dataset; the results are delivered in so-called **batch views**
- **Serving Layer** makes batch views accessible to user queries

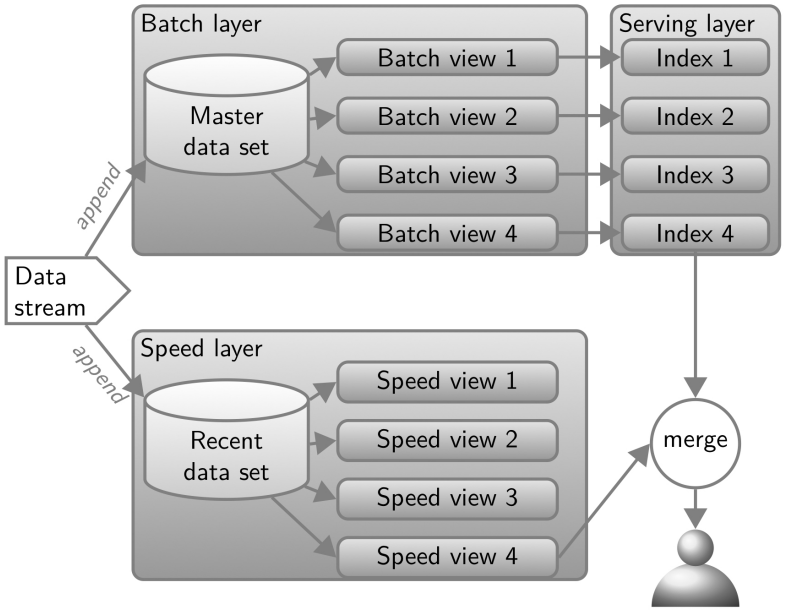


Figure 11.2: Lambda architecture

11.3 Multi-Model Databases

Relying on different storage backends increases the overall complexity of the system and raises different problems, so it might be advantageous to use a database system that stores data in a single store but provides access to the data with different APIs.

> Databases offering this feature have been termed **multi-model** databases. They either support different data models directly inside the database engine or they offer layers for additional data models on top of a single-model engine

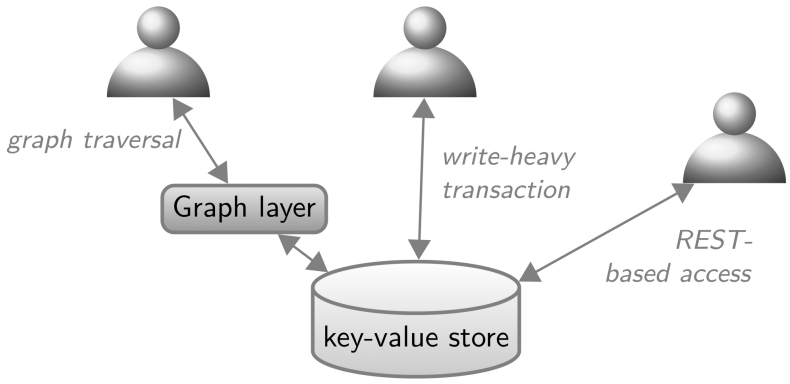


Figure 11.3: A multi-model database

Several advantages come along with this single-database multi-model approach:

- *Reduced database administration:* maintaining a single database is easier
- *Reduced user administration:* only one level of user management is necessary

- *Integrated low-level components:* low-level database components can be shared between the different data models in a multi-model database
- *Improved consistency:* it is a lot easier to ensure
- *Reliability and fault tolerance:* backup just has to be set up for a single database
- *Scalability:* data partitioning and data locality can best be configured in a single database system
- *Easier application development:* programming efforts regarding database administration data models and query languages can focus on a single database system

Part II

Document Databases

Chapter 12

MongoDB

12.1 Document

MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON.

12.1.1 Document Structure

The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. Field names are strings. Documents have the following restrictions on field names:

- The field name *_id* is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array. If the *_id* contains subfields, the subfield names cannot begin with a ($) symbol.
- Field names cannot contain the null character.
- The server permits storage of field names that contain dots (.) and dollar signs ($)
-

12.1.2 Dot Notation

MongoDB uses the dot notation to access the elements of an array and to access the fields of an embedded document. To specify or access a field of an embedded document with dot notation, concatenate the embedded document name with the dot (.) and the field name, and enclose in quotes

12.1.3 Document Limitations

Documents have the following attributes:

- **Document Size Limit:** The maximum BSON document size is 16 megabytes.
- **Document Field Order:** Unlike JavaScript objects, the fields in a BSON document are ordered.

- **The *_id* Field:** In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field.

12.1.4 Other Uses of the Document Structure

- **Query Filter Documents:** Query filter documents specify the conditions that determine which records to select for read, update, and delete operations. You can use $< field >:< value >$ expressions to specify the equality condition and query operator expressions
- **Update Specification Documents:** Update specification documents use update operators to specify the data modifications to perform on specific fields during an update operation.
- **Index Specification Documents:** Index specification documents define the field to index and the index type

12.2 Databases and Collections

MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections. A database stores one or more collections of documents.

12.2.1 Database

In MongoDB, databases hold one or more collections of documents. If a database does not exist, MongoDB creates the database when you first store data for that database.

12.2.2 Collections

MongoDB stores documents in collections. Collections are analogous to tables in relational databases.

- **Create a Collection:** If a collection does not exist, MongoDB creates the collection when you first store data for that collection.
- **Explicit Creation:** MongoDB provides the db.createCollection() method to explicitly create a collection with various options, such as setting the maximum size or the documentation validation rules.
- **Document Validation:** By default, a collection does not require its documents to have the same schema; i.e. the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- **Modifying Document Structure:** To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

- **Unique Identifiers:** Collections are assigned an immutable UUID. The collection UUID remains the same across all members of a replica set and shards in a sharded cluster.

12.3 MongoDB Query API

A document in MongoDB is a data structure composed of field and value pairs. Documents are stored as BSON which is the binary representation of JSON. This low level of abstraction helps you develop quicker and reduces the efforts around querying and data modeling. The document model provides several advantages, including:

- Documents correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Flexible schema. Documents do not need to have the same set of fields and the data type for a field can differ across documents within a collection.

12.4 MongoDB CRUD Operations

https://www.mongodb.com/docs/manual/crud/

12.5 Aggregation Pipeline

An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.
- The same stage can appear multiple times in the pipeline with these stage exceptions: $\$out$, $\$merge$, and $\$geoNear$
- To calculate averages and perform other calculations in a stage, use aggregation expressions that specify aggregation operators. You will learn more about aggregation expressions in the next section.

12.5.1 Aggregation Pipeline Expressions

Some aggregation pipeline stages accept an aggregation expression, which:

- Specifies the transformation to apply to the current stage's input documents.
- Transform the documents in memory.
- Can specify aggregation expression operators to calculate values.
- Can contain additional nested aggregation expressions.

12.6 Replication

A replica set in MongoDB is a group of mongod processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments.

12.6.1 Redundancy and Data Availability

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centers can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

12.6.2 Replication in MongoDB

- A replica set is a group of mongod instances that maintain the same data set.
- A replica set contains several data bearing nodes and optionally one arbiter node.
- Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes.
  - The **primary node** receives all write operations.
  - The **secondaries** replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set. If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.

12.6.3 Asynchronous Replication

Secondaries replicate the primary's oplog and apply the operations to their data sets asynchronously. By having the secondaries' data sets reflect the primary's data set, the replica set can continue to function despite the failure of one or more members.

Replication Lag and Flow Control

Replication lag refers to the amount of time that it takes to copy (i.e. replicate) a write operation on the primary to a secondary. Some small delay period may be acceptable, but significant problems emerge as replication lag grows, including building cache pressure on the primary.

12.6.4 Automatic Failover

When a primary does not communicate with the other members of the set for more than the configured electionTimeoutMillis period, an eligible secondary calls for an election to nominate itself as the new primary. The cluster attempts to complete the election of a new primary and resume normal operations.

The replica set cannot process write operations until the election completes successfully. The replica set can continue to serve read queries if such queries are configured to run on secondaries while the primary is offline.

Lowering the electionTimeoutMillis replication configuration option from the default 10000 (10 seconds) can result in faster detection of primary failure. However, the cluster may call elections more frequently due to factors such as temporary network latency even if the primary is otherwise healthy.

12.7 Read Operations

By default, clients read from the primary [1]; however, clients can specify a read preference to send read operations to secondaries.

- **Asynchronous replication:** to secondaries means that reads from secondaries may return data that does not reflect the state of the data on the primary.
- **Multi-document transactions:** hat contain read operations must use read preference primary. All operations in a given transaction must route to the same member.

12.7.1 Data Visibility

- Depending on the read concern, clients can see the results of writes before the writes are durable.
- For operations in a multi-document transaction, when a transaction commits, all data changes made in the transaction are saved and visible outside the transaction. That is, a transaction will not commit some of its changes while rolling back others.
- Until a transaction commits, the data changes made in the transaction are not visible outside the transaction.

12.7.2 Mirrored Reads

Mirrored reads reduce the impact of primary elections following an outage or planned maintenance. After a failover in a replica set, the secondary that takes over as the new primary updates its cache as new queries come in.

12.7.3 Transactions

Multi-document transactions that contain read operations must use read preference primary. All operations in a given transaction must route to the same member.

12.8 Sharding

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations. There are two methods for addressing system growth: vertical and horizontal scaling.

- **Vertical Scaling:** involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.
- **Horizontal Scaling:** involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required.
  - Better efficiency than a single high-speed high-capacity server
  - Expanding the capacity of the deployment only requires adding additional servers, so lower cost

12.8.1 Sharded Cluster

A MongoDB sharded cluster consists of the following components:

- **shard:** Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.
- **mongos:** The mongos acts as a query router, providing an interface between client applications and the sharded cluster.
- **config servers:** Config servers store metadata and configuration settings for the cluster.

12.8.2 Shard Keys

MongoDB uses the shard key to distribute the collection's documents across shards. The shard key consists of a field or multiple fields in the documents.

- You select the shard key when sharding a collection.
- A document's shard key value determines its distribution across the shards.

Shard Key Index

To shard a populated collection, the collection must have an index that starts with the shard key.

Shard Key Strategy

The choice of shard key affects the performance, efficiency, and scalability of a sharded cluster. A cluster with the best possible hardware and infrastructure can be bottlenecked by the choice of shard key.

12.8.3 Advantages of Sharding

- **Reads / Writes**: MongoDB distributes the read and write workload across the shards in the sharded cluster, allowing each shard to process a subset of cluster operations.
- **Storage Capacity:** Sharding distributes data across the shards in the cluster, allowing each shard to contain a subset of the total cluster data.
- **High Availability:** The deployment of config servers and shards as replica sets provide increased availability.

12.8.4 Considerations Before Sharding

- Sharded cluster infrastructure requirements and complexity require careful planning, execution, and maintenance.
- Once a collection has been sharded, MongoDB provides no method to unshard a sharded collection.
- While you can reshard your collection later, it is important to carefully consider your shard key choice to avoid scalability and perfomance issues.

12.8.5 Sharded and Non-Sharded Collections

A database can have a mixture of sharded and unsharded collections. Sharded collections are partitioned and distributed across the shards in the cluster. Unsharded collections are stored on a primary shard. Each database has its own primary shard.

12.8.6 Connecting to a Sharded Cluster

You must connect to a mongos router to interact with any collection in the sharded cluster. This includes sharded and unsharded collections.

12.8.7 Sharding Strategy

- **Hashed Sharding:** involves computing a hash of the shard key field's value. Each chunk is then assigned a range based on the hashed shard key values.
- **Ranged Sharding:** involves dividing data into ranges based on the shard key values. Each chunk is then assigned a range based on the shard key values.

12.8.8 Zones in Sharded Clusters

Zones can help improve the locality of data for sharded clusters that span multiple data centers.

- In sharded clusters, you can create zones of sharded data based on the shard key.
- You can associate each zone with one or more shards in the cluster.

- Each zone covers one or more ranges of shard key values.
- You must use fields contained in the shard key when defining a new range for a zone to cover.

12.8.9 Transactions

Starting in MongoDB 4.2, with the introduction of distributed transactions, multi-document transactions are available on sharded clusters.

- Until a transaction commits, the data changes made in the transaction are not visible outside the transaction.
- However, when a transaction writes to multiple shards, not all outside read operations need to wait for the result of the committed transaction to be visible across the shards.

Chapter 13

Cypher

13.1 Cypher path matching

Neo4j Cypher makes use of relationship isomorphism for path matching and is a very effective way of reducing the result set size and preventing infinite traversals.

Part III

DBMS Internals

Chapter 14

DBMS Functionalities and Architecture

14.1 Overview of a DBMS

A database (DB) is a collection of homogeneous sets of data, with relationships defined among them, stored in a permanent memory and used by means of a Data Base Management System (DBMS), a piece of software that provides the following key features:

- A language for the database *schema* definition. A collection of definitions which describe the data structure, the restrictions on allowable values of the data, and the relationships among data sets
- The data structures for the storage and efficient retrieval of large amounts of data in permanent memory
- The data structures for the storage and efficient retrieval of large amounts of data in permanent memory. Or to rapidly retrieve interesting subsets of the data from a specification of their features
- A transactions mechanism to protect data from hardware and software malfunctions and unwanted interference during concurrent access by multiple users

14.2 DBMS Architecture

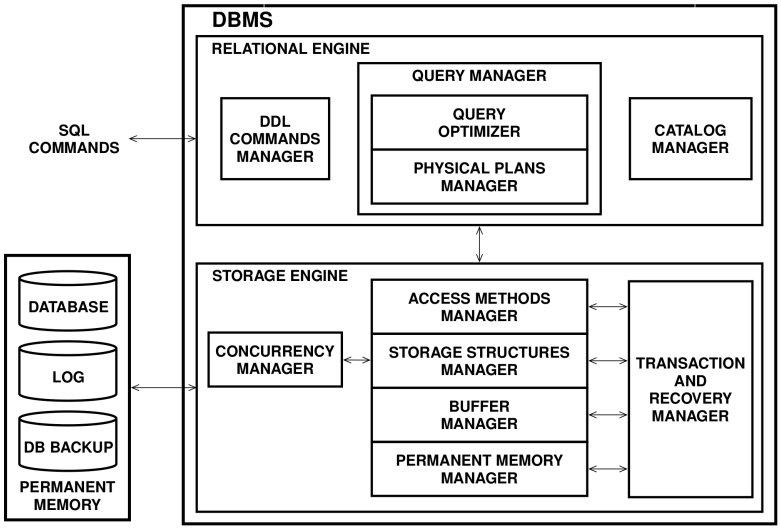


Figure 14.1: Architecture of a DBMS

- The **Storage Engine** which includes:
  - The *Permanent Memory Manager* which manages the page allocation and de-allocation on disk storage
  - The *Buffer Manager* which manages the transfer of data pages between the permanent memory and the main memory
  - The *Storage Structure Manager* which manages the data structures to store and retrieve data efficiently
  - The *Access Methods Manager* which provides the storage engine operators to create and destroy databases, files, indexes, and the data access methods for table scan, and index scan
  - The *Transaction and Recovery Manager* which ensures that the database's consistency is maintained despite transaction and system failures
  - The *Concurrency Manager* which ensures that there is no conflict between concurrent access to the database.
- The **Relational Engine** which includes:
  - The *Data Definition Language (DLL) Manager* which processes a user's database schema definition
  - The *Query Manager* which processes a user's query by transforming it in to an equivalent but more efficient form
  - The *Catalog Manager* which manages special data, called metadata, about the schemas of the existing databases, and security and authorization

Let us briefly examine the modules that will be considered in the following chapters:

- The **Permanent Memory Manager** provides a vision of the memory as a set of databases each consisting of a set of files of physical pages of fixed size
- The **Buffer Manager** the execution cost of some queries can be reduced using a buffer capable of containing many pages, so that, while executing the queries, if there are repeated accesses to the same page, the likelihood that the desired page is already in memory increases
- The **Storage Structure Manager** provides the other system levels with a view of the permanent data organized into collections of records and indexes
- The **Access Methods Manager** provides a vision of permanent data organized in collections of records accessible one after the other in the order in which are stored, or through indexes, abstracting from their physical organization
- The **Transaction and Recovery Manager** provides the other system levels with a vision of the permanent memory as a set of pages in temporary memory without regard to failures
- The **Concurrency Manager** provides the other system levels with a vision of permanent memory as a set pages in memory without regard to concurrent access, thus ensuring that the concurrent execution of several transactions takes place as if they were executed one after the other
- The **Query Manager** provides a vision of permanent data as a set of relational tables on which a user operates with SQL commands

14.3 The JRS System

The implementation of the relational DBMS modules will be discussed first in general and then with respect to the solutions adopted for the system JRS (Java Relational System), developed in Java at the Department of Computer Science, University of Pisa.

It is a simple system with all the most important components of a RDBMS, and its Java code can explored to learn about how such a system works.

Chapter 15

Permanent Memory and Buffer Management

The first problem to be solved in implementing a DBMS is to **provide a level of abstraction** of the permanent memory that makes the *other modules of the system* **independent** of its characteristics and of those of the storage system. All of this is achieved with the **Permanent Memory Manager** and **Buffer Manager**.

15.1 Permanent Memory

The memory managed by a DBMS is usually organized in a two-level hierarchy: the **temporary memory** (or main) and the **permanent memory** (or secondary).

- **Main/Temporary memory**
  - Electronic
  - Fast access to data
  - Small capacity
  - Volatile
  - Expensive
- **Permanent memory with magnetic disks**
  - Slow access to data
  - Large capacity
  - Non volatile
  - Cheap
- **Permanent memory with NAND flash memory**
  - Relatively fast access to data
  - Medium capacity
  - Non volatile (Persistent)
  - Relatively expensive

The flash memory, with the decrease in the cost and the increase in their capacity, is destined to establish itself for personal computers as an alternative to magnetic disks. However they poses new challenges to the implementation of DBMSs, due to timing characteristics.

	Access Time		
Memory	**Read**	**Write**	**Erase**
Magnetic Disk	12,7 ms (2 KB)	13,7 ms (2 KB)	
NAND Flash	80 μs (2 KB)	200 μs (2 KB)	1,5 ms (128 KB)
RAM	22,5 ns	22,5 ns	

Figure 15.1: Characteristics of the three types of memory

- The reading and writing of data are faster than those of magnetic disks, but the rewriting of data is problematic because the data must be deleted first
- The erasing of data concerns several blocks called **memory unit** which should all be read to be deleted
- This type of memory becomes unreliable after 100 000 cycles of cancellations/rewrites

15.1.1 Parentheses on Magnetic Disk

- A magnetic disk is composed of a pile of p **platters** with concentric rings called **tracks** used to store data
- A **track** is the part of the disk that can be used without moving the read head and it is divided in *sectors* of the same size, which are the smallest unit of transfer allowed by the hardware and cannot be changed
- A **track** is logically divided in *blocks* of fixed with a multiple of sector size
- The disk driver has an array of **disk heads**, one per recorded surface. Each head is fixed on a movable arm that displaces the head horizontally on the entire surface of a disk
- A **cylinder** is the set of tracks of the surfaces of the disks that can be used without moving the heads

The time to read or write a block, called the **access time**, has the following components:

- The **seek time** t_s**:** time needed to position the disk heads over the cylinder containing the desired block
- The **rotational latency** t_r**:** the time spent waiting for the desired block to appear under the read/write head

- The **transfer time** t_b**:** the time to read or write the data in the block once the head is positioned

The transfer time fora single block in the temporary memory is then $(t_s + t_r + t_b)$, while the transfer time for k contiguous blocks is $(t_s + t_r + k \times t_b)$.

Moreover since *seek time + latency* is much greater than *transfer time*, it is important:

- To transfer greater blocks rather than small ones
- Makes consecutive reads on consecutive blocks on the same cylinder

15.2 Permanent Memory Manager

- The Permanent Memory Manager takes care of the *allocation* and *de-allocation* of pages within a database, and performs *reads* and *writes* of pages to and from the disk
- It provides an **abstraction of the permanent memory** in terms of a set of databases each made of a set of files with page-sized blocks of bytes, called **physical pages**
  - The physical pages of a file are numbered consecutively starting from zero
  - Can grow dynamically, limited to the permanent memory available space
  - When it is transferred to the main memory it is called a **page**

15.3 Buffer Manager

- Its role is to make pages from the permanent memory available to transactions in the main memory
- It is responsible to to allow transactions to get the pages they need, while minimizing disk access operations, by implementing a page replacement strategy

The performance of operations on a database depends on the number of pages transferred to temporary memory. The cost of some operations may be reduced by using a buffer capable of containing many pages.

The buffer manager uses the following structures to carry out its tasks:

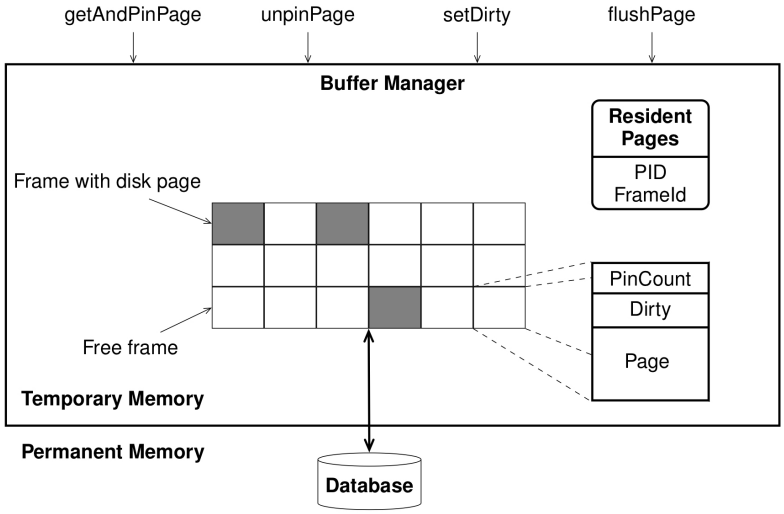


Figure 15.2: The Buffer Manager

- The **buffer pool:**
  - It is an array of frames each one containing a copy of a permanent memory page
  - It has a fixed size, so when there are no free frames, in order to copy a new page from the permanent memory an appropriate algorithm is used in order to free a frame
  - To manage it in a frame are also stored two variables:
    - *pin count:* initially *false* and it stores the number of times that the page currently in the frame has been requested but not released
    - *dirty bit:* initially 0 and it indicate whether the page has been modified
- A hash **resident pages** tables, called **directory:** used to know if a permanent memory page, with a given page identifier PID, is in the buffer pool, and which frame contains it

The buffer manager provides the following primitives to use the pages in the buffer pool:

- *getAndPinPage(P):* iff a frame contains the requested page, it increments the pin count of that frame and returns the page identifier to the requester. And if the requested page is not in the buffer pool, it is brought in as follows:

  1. A free frame is chosen according to the buffer management's replacement policy, most often *Least Recently Used (LRU)*. If the frame chosen for replacement is dirty, the buffer manager flush it. If there are no free frames an exception is raised
  2. The requested page is read into the frame chosen for replacement and *pinned*
  3. The resident pages table is updated, to delete the entry for the old page and insert an entry for the new page

- *setDirty(P):* if the requestor modifies a page, it asks the buffer manager to set the dirty bit of the frame

- *unpinPage(P):*w hen the requestor of a page releases the page no longer needed, it asks the buffer manager to unpin it

- *flushPage(P)* the requestor of a page asks the buffer manager to write the page to the permanent memory if it is dirty

Chapter 16

Heap and Sequential Organizations

16.1 Storing Collections of Records

- A database is primarily made of **tables of records**, each one implemented by the *Storage Structures Manager* as a **file of pages** provided by the *Permanent Memory Manager*
- Pages are assumed to be of a fixed size and to contain several records
- The **unit of cost** for data access is a *page access*
- The most important *type of file* is the **heap file**, which stores records in no particular order, and provides a record at a time interface for accessing, inserting and deleting records

16.1.1 Record Structure

- Each record consists of one or more *attributes* and contains several additional bytes, called *record header* useful for record management
- *Record header* generally contain:
  - Information on the length of the record
  - The number of attributes
  - Whether the record has been deleted
  - The name of the file to which it belongs

We will assume that records are not larger than a page and that the values of the attributes are stored according to one of the strategies shown:

A fixed-length record

Attribute	Position	Value type	Value
Name	1	char(10)	Rossi
StudentCode	2	char(6)	456
City	3	char(2)	MI
BirthYear	4	int(2)	68

Total number of characters = 20
Attribute values are separated:
a) by position Rossi␣␣␣␣␣456␣␣␣MI68
b) with a separator Rossi@456@MI@68
c) with an index* (1, 6, 9, 11) Rossi456MI68
d) with labels** (6, 1, Rossi)(4, 2, 456)(3, 3, MI)(3, 4, 68)

* The index, placed at the beginning of each record, indicates the beginning of each attribute value.
** Each attribute value starts with a counter that tells how many characters are used to code the position of the attribute and its value.

Figure 16.1: Representation of attribute values

16.1.2 Page Structure

- When a record is stored in the database, it is identified by a **record identifier** or **tuple identifier (RID)**, used like a pointer to record

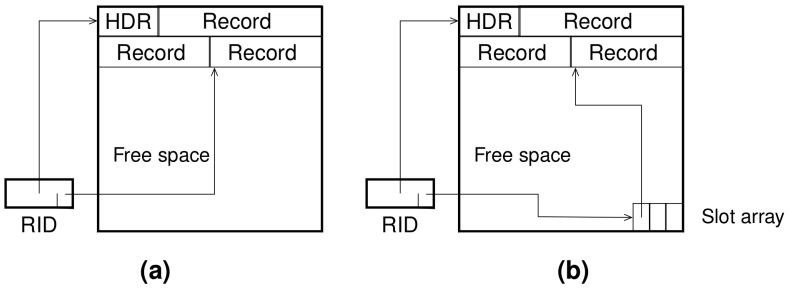


Figure 16.2: Pointers to records

The exact nature of a RID can vary from one system to another:

- We could take its **address (a)**. But this solution is not satisfactory because a record that contains variable-length attributes of type *varchar* are themselves variable-length strings within a page
- RID formed by **two parts (b)** *(Page number, Slot number)*. Where the slot number is an index into an array stored at the end of the page, called **slot array**.

If an updated record moves within its page, the local address in the array only must change, while the RID does not change. If an updated record cannot be stored in the same page because of lack of space, then it will be stored in another page, and the original record will be replaced by a forwarding pointer.

Each page has a **page header** that contains administrative information like:

- The number of free bytes in the page
- The reference at the beginning of the free space
- The reference to the next not empty page of the file

16.1.3 File Pages Management

When a record is inserted into a collection, assuming that the records have size smaller than a page, the file manager proceeds as follows:

- A file page is selected that contains free space for the new record; if the page does not exist, the file is extended with a new page. Let Let P be the address of the page where the record will be stored
- A reference to the beginning of the record is stored in the first free location j of the directory of slots of the page P
- The RID(P, j) is assigned to the record

To implement insertion, the system uses a *table* stored on disk, containing pairs of *(fileName, headerPage)*, where the header page is the first page of the file, and the following alternatives are usually considered:

- The heap file pages are organized as *two double linked list* of pages, those full and those with free space
- In the header is stored a directory of pages and each entry contains the pair (page identifier, the amount of free space on the page). If the directory grows and cannot be stored in the header page, it is organized as a linked list. The information about the amount of free space on a page is used to select a page with enough space to store a record to be inserted

Finally, for reasons of efficiency, the free space existing in different pages is not compacted in one page by moving records. Therefore, it may happen that, due to a lack of available pages, it is not possible to assign a new free page despite the fact that the overall free space in different pages is greater than the total capacity of a page.

16.2 Cost Model

The most important criteria to evaluate a file organization are the **amount of memory occupied** and the **cost of the basic operations**.

> The most important operation is the search, because the first step for all operations is to check whether a record exists.

We will estimate the value of the following parameters:

- $N_{rec}(R)$: number of record stored in file R
- L_r: record length
- $N_{pag}(R)$: number of pages where the record is stored
- D_{pag}: page size

The operations cost will be estimated by considering only the operations to read and write a file page. The cost of the operations will be expressed in terms of the number of permanent memory accesses.

When, in some instances, we want to emphasize the magnitude of the execution time, we will make the simplifying assumption that the time to perform an operation is a simple function of the number of access operations:

$$ExecutionTime = NoAccesses \times OneAccessAverageTime$$

16.3 Heap Organization

> The simplest way to organize data is to store it in file pages in the insertion order rather than sorted by a key value. The heap organization is the default solution used by DBMSs, and it is adequate for:
>
> - Small collection of records
> - Infrequent key search

16.3.1 Performance Evaluation - (NOT REQUIRED)

- **Memory Requirements:** the *memory occupied* by a heap organization is only that required by the in stored records:

$$N_{pag}(R) = N_{rec}(R) \times L_r / D_{pag}$$

- **Equality Search**: assuming that the distribution of the key values is uniform, the *average search* cost is:

$$C_s = \begin{cases} \lceil \frac{N_{pag}(R)}{2} \rceil, & \text{if the key exists in the file} \\ N_{pag}(R), & \text{if the key does not exist in the file} \end{cases}$$

- **Range Search**: all file pages must be read so it is:

$$N_{pag}(R)$$

- **Insert**: record is inserted at the end of the file, and the cost is 2
- **Delete and Update**: The cost is that of a key search plus the cost of writing back a page:

$$C_s + 1$$

16.4 Sequential Organization

- sequential organization is used for efficient processing of records stored in sequential order, according to the value of a search-key k for each record.
- **Disadvantage:** t it is costly to maintain the sequential order when new records are inserted in full pages.

16.4.1 Performance Evaluation - (NOT REQUIRED)

Type	Memory	Single value search	Interval search	Insertion	Deletion
Serial	$N_{pag}(R)$	$\lceil N_{pag}(R)/2 \rceil$	$N_{pag}(R)$	2	C_s+1
Sequential	$N_{pag}(R)$	$\lceil \log_2 N_{pag}(R) \rceil$	$C_s+1+\lceil f_s \times N_{Pag}(R) \rceil$	$C_s+1+N_{Pag}(R)$	C_s+1

- **Memory Requirements:** If record insertions are not allowed, the organization requires the same memory as a heap organization.
- **Equality Search**: The cost of a search by a key value both when the value exists in the file and when the value does not exist, is:

$$\lceil N_{pag}(R) \rceil$$

If the data is stored in *consecutive pages*, then a binary search has the cost:

$$\lceil lgN_{pag}(R) \rceil$$

- **Range Search**:
  - A search by the *key* k in the range $(k_1 \leq k \leq k_2)$
  - Assuming keys numerical uniformly distributed in the range (k_{min}, k_{max})
  - The *ratio* $s_f = (k_2 - k_1)/(k_{max} - k_{min})$ called **selectivity factor** is an estimator of the fraction of pages occupied by the records
  - And the cost of this operation is:

$$C_s = \rceil lgN_{pag}(R) \lceil + \rceil s_f \times N_{pag}(R) \lceil - 1$$

- **Insert**:
  - If the record must be inserted in a page not full

    $$C_s + 1$$

  - If all the pages are full, the cost is estimated by assuming that the record must be inserted in the *middle of the file*, so we have to move half of pages $N_{pag(R)}$ thus the total cost is:

    $$C_s + N_{pag}(R) + 1$$

- **Delete and Update**: $C_s + 1$ if the update does not change the key on which data is sorted.

16.5 Comparison of Costs

This table compares costs for heap and sequential organizations in consecutive pages, with C_s as the search cost of a key value present in the file.

- **Heap organization**
  - *Advantage:* good performance for insertion operations
  - *Disadvantage:* bad performances for range queries and for equality search
- **Sequential organization**
  - *Advantage:* good performance for search operations
  - *Disadvantage:* bad performances for insertion operations

16.6 External Sorting

A frequent operation in a database system is sorting a collection of records. Sorting a file is a different process from sorting a table in the temporary memory, because the number of record is usually too large to be completely stored in the memory available. For this reason, the classic algorithms for sorting tables are not applicable to this problem and an external sorting algorithm is used.

Let $N_{pag}(R)$ be the number of file pages, and B the buffer pages available, the classical *external sorting algorithm*, called **merge-sort** consist in two phases:

- The **sort phase:**
  - B file pages are read into the *buffer*, sorted, and written to the disk
  - This creates $n = N_{pag}(R)/B$ sorted subset of records called *runs*, numbered from 1 to n
- The **merge phase:**
  - It consists of *multiple merge passes*

- In each, $Z = B - 1$ runs are merged using the remaining buffer page for output.
- At the end the number of runs are $n = \lceil n/Z \rceil$

The parameter Z is called the **merge order** and $Z + 1$ buffer pages are needed to proceed with a **Z-Merge**

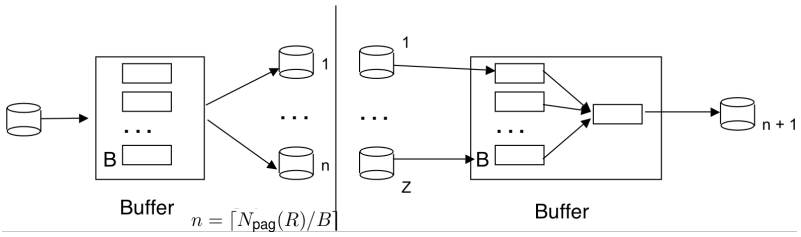


Figure 16.3: Sort-Merge

Example

Let us show how to sort the file A_0 containing 12 pages, with file and buffer pages capacity of 2 records, $B = 3$ and 2-merge passes:

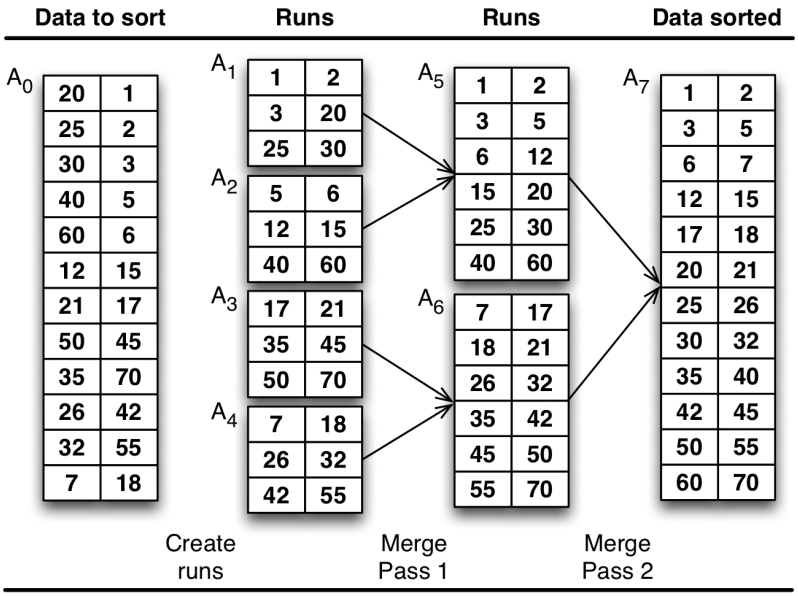


Figure 16.4: Example of Sort-Merge

16.6.1 Performance Evaluation - (NOT REQUIRED)

- Suppose that B buffer pages are available
- The external sorting cost is evaluated in term of number of passes, thus $N_{pag}(R)$ are read in and written out
- The number of passes is the *initial one* to produce the sorted runs, plus the number of the *merge passes*

- The **total cost of the merge-sort algorithm in terms of number of pages:**

$$C_{sort}(R) = SortPhaseCost + MergePhaseCost$$

$$C_{sort}(R) = 2 \times N_{pag}(R) + 2 \times N_{pag}(R) \times NoMergePasses$$

- If $N_{pag}(R) \leq B \times (B-1)$ the data can be sorted with a singe pass, thus the cost becomes:

$$C_{sort}(R) = 4 \times N_{pag}(R)$$

- In general, the number of passes required in the merge phase is a function of the number of file pages $N_{pag}(R)$, the number S of initial runs, and the merge order $Z = B - 1$

- After each merge pass, the maximum length of the runs increases of a factor Z, and so their number becomes:

$$\lceil S/Z \rceil, \lceil S/Z^2 \rceil, \lceil S/Z^3 \rceil ...$$

- The algorithm terminates when a single run is generated, therefore the number of passes required in the merge phase is:

$$k = \lceil \log_Z S \rceil$$

- Therefore, **the total cost of the merge-sort is:**

$$C_{sort}(R) = 2 \times N_{pag}(R) + 2 \times N_{pag}(R) \times \lceil \log_Z S \rceil$$

$$C_{sort}(R) = 2 \times N_{pag}(R) \times (1 + \lceil \log_Z S \rceil)$$

Chapter 17

Hashing Organization

17.1 Table Organizations Based on a Key

The **goal** of a table organization based on a key is to allow the retrieval of a record with a specified key value in as few accesses as possible, 1 being the optimum. To this end, a mapping from the set of keys to the set of records is defined, and can be implemented in two ways:

1. **Primary Organization:** if the file organization determines the way the records are physically stored, and thus retrieved. The mapping from a *key* to the *record* can be implemented as a **hashing technique** or via a **tree structure**

   - The **hash function** h is used to map the key value k to the value $h(k)$ which is used as the address pf the page in which the record is stored
   - The **tree structure** is used to store the record in the leaf nodes

   We have an other distinction in primary memory:

   - **PO is static** if once created *for a known table size,* its performance **degrades** as the table grows
   - **PO is dynamic** if once created *for a known table size,*, it gradually **evolves** as records are added or deleted

2. Otherwise is defined as **secondary organization**. Here the mapping from a *key* to the *record* is implemented with the *tabular method*. It is implemented as an index like follow:

   - An index I on a key K of a set of records R is a sorted table $I(K, RID)$ on K
   - An element of the index is a pair (k_i, r_i)
     - k_i value for a record
     - r_i is a reference (RID) to the corresponding record

17.2 Static Hashing Organization

- This is the oldest and simplest method for a primary table organization based on a key
- We assume that the records have the same and fixed size, and that the key k has a type integer
- The N records of a table R are stored in an area, called *primary area*, divided into M *buckets*, that consist of one page with a capacity of c records
- A record is inserted in a page whose address is obtained by applying a hashing function H
- Different keys can be mapped to the same address and when a page is full and there is another key mapped to it, we have an "overflow"
- In general the possible keys are much more than the records

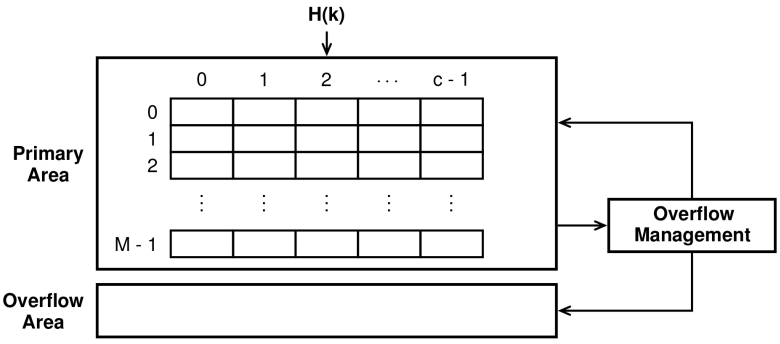


Figure 17.1: Static hashing organization

The design of a static hashing organization requires the specification of the following parameters:

- **Hash Functions**
  - The hash function H is uniform if all the addresses produced are **uniformly distributed** in the interval $(0, M-1)$
  - Two keys k_1 and k_2 are synonyms if $H(k_1) = H(k_2)$, that is if they produce a **collision**
  - If the number of collisions is greater than the page capacity, there is an **overflow**. The overflows, during the search, increases the cost of the operation
  - When it is well designed (80% page occupancy), we can assume that there are no overflows and so a record is retrieved with 1 page access.
  - The typical hash function is the "division method" (Mp is a prime number):

$$H(k) = k \bmod M_p$$

- **Overflow Manager**
  - *Open Overflow* performs a primary area linear search to find the first available page to insert the overflow record. When the last page has been searched, the process starts back
  - *Chained overflow* inserts the overflow record in another page of a different overflow area
- **Loading Factor** computed as $d = N/(M \times c)$
  - High loading factor reduces memory size and increases the cost of operation
  - Influence of loading factor on average search cost
- **Page Capacity**
  - Describe the point in which we start to have overflow, thus it is better to have large capacity
  - If page capacity increases ,the percentage of overflows decreases

17.2.1 Performance Evaluation

A static hashing organization has excellent performance as long as there are no overflows to manage. Overflows quickly degrade performance and so a reorganization must be performed of the hash structure.

- **Search for single key:** 1 page access
- **Range search:** cannot be performed
- **Insertion:** usually cost 1 page access in case of overflow it will cost 2
- **Deletion** Cost of searching plus writing the page , it costs 2 page access

To reduce the cost of data loading, and to improve then performance, the operation proceeds as follow:

1. Build a temporary file of pairs: (Record with key $K_i, H(K_i)$)
2. Sort the file on $H(K)$
3. Load the records in the hash files, in two passes:
   (a) Load the records which do not overflow
   (b) Then load the records which do overflow

The **reorganization** of the hash file is obtained by copying the records in an auxiliary file and then doing a loading. It is needed when:

- When the overflow percentage is high
- When the file has a high volatility, to reuse the space of logical deletions

17.3 Dynamic Hashing Organizations

Several dynamic hashing organizations have been proposed to avoid the reorganization which is necessary in static hashing organizations. Dynamic hashing can be summarize with two groups:

- **With auxiliary data structures:** *virtual hash, extensible hash, dynamic hash*
- **Without auxiliary data structures:** *linear hash, spiral hash*

17.3.1 Virtual hashing

Virtual hashing works as follow:

- The data area contains initially M contiguous pages with a capacity of c records. A page is identified by its address, a number between 0 and $M-1$.
- A bit vector β is used to indicate with a 1 which page of the data area contains at least a record.
- Initially a hashing function H_0 is used in order to map each key value k to an address $m = H_0(k)$, between 0 and $M-1$ where the record with the key k should be stored. If an overflow is generated then:
  - The data area is doubled
  - The hashing function H_0 is replaced by the new hashing function H_1 that produces page addresses between 0 and $2M-1$
  - The hashing function H_1 is applied to k and all the records of the original overflowing page m to distribute the records between m and a new page m' in the new half of the table.

This method requires the use of a hashing functions series $H0, H1, H2, ..., Hr$; in general, Hr produces a page address between 0 and $2^r M - 1$. The index of the hashing function H is the number of times that the data area has been doubled.

The function suggested by Litwin is:

$$H_r(k) = k \bmod (2^r \times M)$$

Where

- k is the key of a record to be retrieved
- r is the number of doubling of the data area.

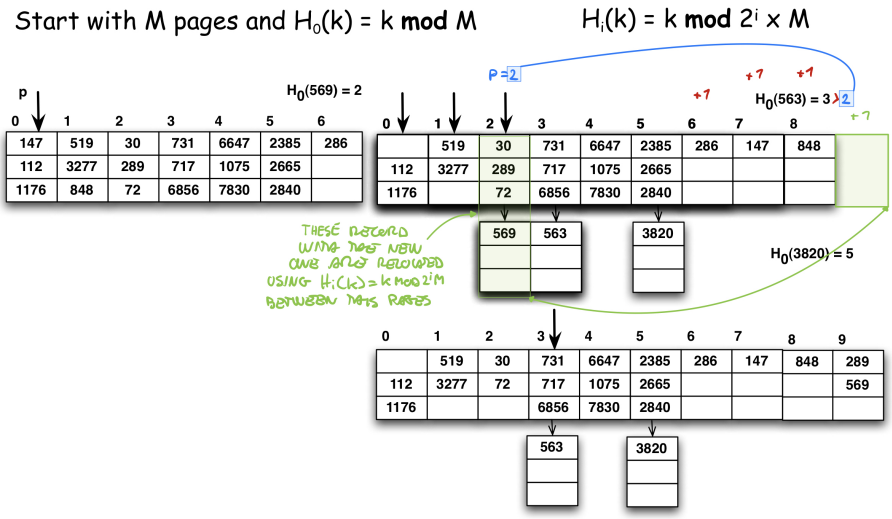


Figure 17.2: Example of Virtual Hashing

```
function PageSearch(r, k: integer): integer
    begin
        if r < 0
            then write "The key does not exist";
            else if B(H_r(k)) = 1
                then PageSearch := H_r(k)
                else PageSearch := PageSearch(r − 1, k)
    end;
```

Figure 17.3: Search operation

17.3.2 Linear Hashing

- The idea of this method is again to increase the number of data pages as soon as a page overflows
- The page which is split is not the one that flows over, but the page pointed by the current pointer p, initialized to the first page ($p = 0$) and incremented by 1 each time a page is split.

The process is like so:

1. M pages are allocated and the hash function is $H_0(k) = k \bmod M$
2. When there is an overflow from a page with address $m \geq p$ an overflow chain is maintained for the page m, but a new page is added
3. The records in page p, and possible overflows from this page, are distributed between the page p and the new page using the hash function $H_1(k) = k \bmod 2M$ which generates addresses between 0 and $(2M - 1)$

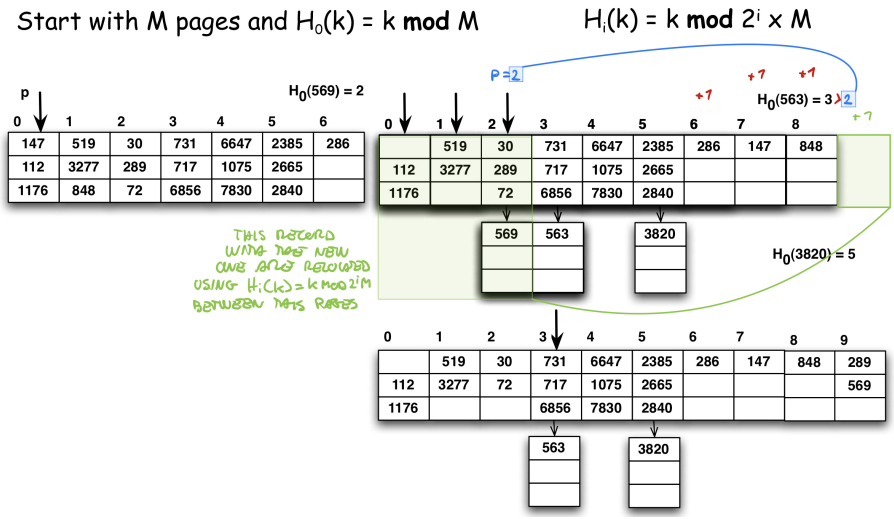


Figure 17.4: Example of Linear Hashng

Chapter 18

Dynamic Tree-Structure Organizations

18.1 B-trees

A *B-tree* of order m ($m \geq 3$) is an m-way search tree that is either empty or of height $h \geq 1$ and it satisfy the following proprieties:

1. Each node contains at most $m - 1$ keys
2. Each node, except the root, contains at least $\lceil m/2 \rceil - 1$ keys
3. A node is either a leaf node or has $j + 1$ children, where j is the number of keys of the node
4. All leaves appear on same level
5. Each node has the following structure:

$$[p_0, k_1*, p_1, k_2*, p_2, ..., k_j*, p_j]$$

where:

- The keys are sorted: $k_1 < ... < k_j$
- p_i is a pointer to another node of the tree structure, and is undefined in the leaves

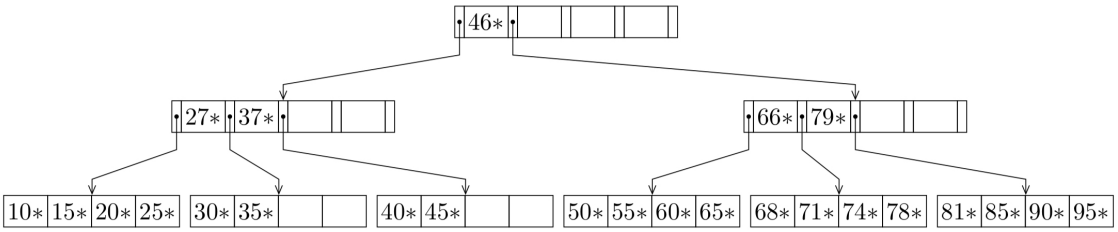


Figure 18.1: B-tree example

18.1.1 Operations

Search

The search start at the root node and if the key is not in the root and $h > 1$ the search continue deeply in the tree like so:

1. If $k_i < k < k_{i+1} \quad 1 \leq i \leq m$ then the search continues in the subtree p_i
2. If $k_m < k$, then the search continues in the subtree p_m
3. If $k < k_1$ the search continues in the subtree p_0

Insertion

- First, a search is made for the leaf node which should contain the key k.
- An unsuccessful search determines the leaf node Q_1 where k should be inserted
  - If the node Q_1 contains less than $m - 1$ keys, then k is inserted and the operation terminates
  - Otherwise, if Q_1 is **full**, it will be split into two nodes, with the *first half* of the m keys that remain in the old node Q_1, the *second half* of the keys that go into a new adjacent node Q_2, and the median key, together with the pointer to Q_2, that is inserted into the father node Q of $Q1$

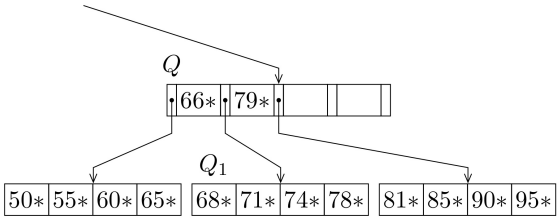


Figure 18.2: Insertion key 70 - Step 1

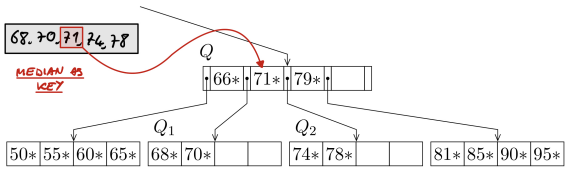


Figure 18.3: Insertion key 70 - Step 2

Deletion

The deletion is always effected on the leaf nodes. Furthermore, after a deletion if the leaf node p has less than $\lceil m/2 \rceil - 1$ keys, it has to be regrouped with an adjacent brother node using one of the following techniques:

- **Merging:** the node p is *merged* with one of its adjacent brother nodes which contains $\lceil m/2 \rceil - 1$ elements
- **Rotation:** when the merging of the node p with one of its adjacent brothers is not possible, then the *rotation* technique is applied

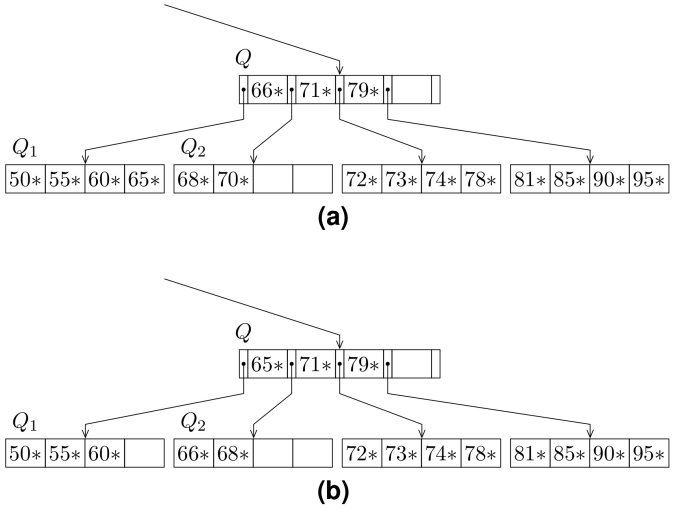


Figure 18.4: A rotation example

18.2 Performance Evaluation

Let us evaluate the costs of the operations expressed in terms of the number of nodes to read and write.

18.2.1 Search

The height determines the cost of simple key search since $1 \leq C \leq h$, and this relation holds:

$$log_m(N+1) \leq h \leq 1 + log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)$$

18.2.2 Range Search

B–trees are very good for equality searches, but not to retrieve data sequentially or for range searches, because they require multiple tree traversals.

18.2.3 Insertion

An insertion is made in a leaf node. If the node is not full, the new key is inserted keeping the node's keys sorted. The cost is h reads and 1 write

18.2.4 Deletion

The cost of the operation is estimated by considering three cases;

1. If the key is **in a leaf** and the **merging and rotation is not required**, the cost is h and 1 write

2. If the key is **in a node** and the **merging and rotation is not required**, the cost is h and 2 write

3. The worst case is when for all the nodes of the path from the root to the node, except for the first two, the **merging** operation is required, and for the child of the root a **rotation** operation is required, The cost is $2h-1$ reads and $h+1$ writes

18.3 B^+-trees

A B^+–tree is a well known B–tree variant to enhance search performance especially for range queries. In a B^+–tree all the records, denoted as $k*$, are stored sorted in the leaf nodes, organized int a doubly linked list.

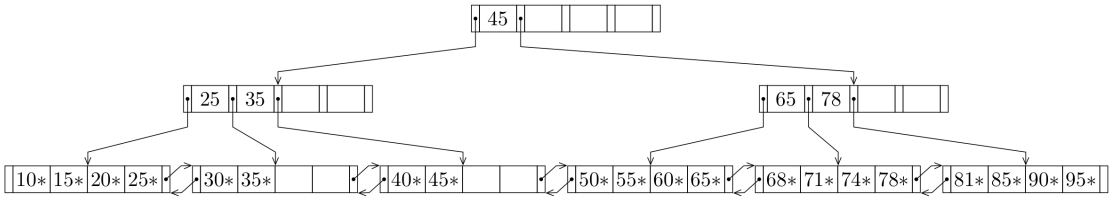


Figure 18.5: B+tree Example

B^+–tree are different from the previous B-tree for the following reasons:

1. A key search requires always the same number of accesses equal to the B^+–tree height

2. Key search is faster, because the records are stored in the leaf nodes, and only the keys are stored in the non-leaf nodes of the tree

3. When a leaf node F is split into F_1 and F_2, a copy of the highest key in F_1 is inserted in the father node of F, while when an internal node I is split the median key in I is moved in the father node, as in a B–tree.

4. When a record with the key k_i is deleted, the k_i* is deleted in the leaf F, and if k_i is used in a father node because it was the highest key in F, it is not necessary to replace it with the new highest key in F

18.4 Index Organization

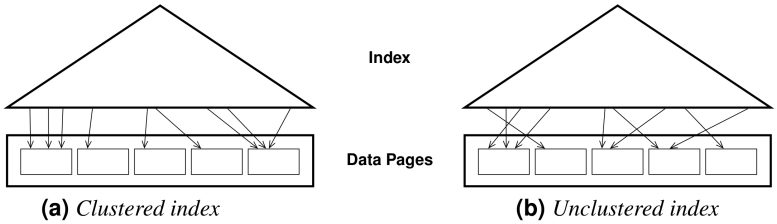


Figure 18.6: Types of indexes

To support fast retrieval of records in a table using different keys, two types of indexes can be used:

- If the table is stored with a *heap organization*, an index is defined for each key, and the elements of the indexes are pairs (k_i, rid_i) where k_i is a *key value* for a record and rid_i is the record identifier

  > A **clustered index** on a key K of a table is created by first sorting the table on the values of the index key K. If new records are inserted into the table after the clustered index is created, the efficiency of the index decreases. To overcome this problem we specify that a small fraction of each data page is left empty for future insertions.

  A clustered index is particularly useful for a range key search.

- If the table is stored with a *dynamic primary organization* using the "primary key", indexes are defined for the other keys, and the element of the indexes are pairs (k_i, pk_i) where k_i is the value for a record and pk_i is the primary key value of the corresponding value

There are two additional types of indexes:

1. An index on a key is called **dense** if the number of its entries is equal to the number of records stored into a separated data file

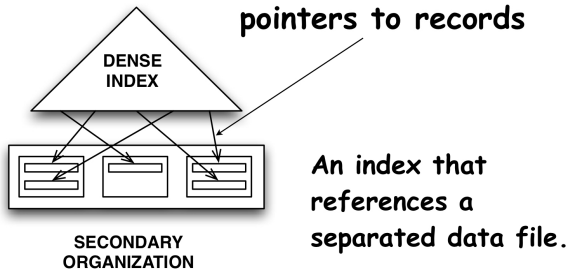


Figure 18.7: Dense index

2. The term **sparse index** is used for the part of the tree structure used to locate the data records stored sorted in the leaves of a B+-tree. It is called sparse because the number of its entries is less than the number of data records

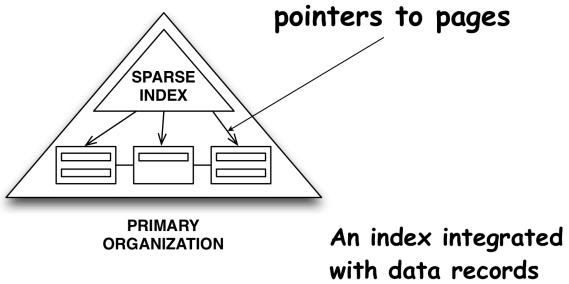


Figure 18.8: Sparse index

18.4.1 Performance Evaluation - (NOT REQUIRED)

Unclustered Index

- **Search Cost:** $C = C_i \times C_d$ = cost for index access plus cost for data access. Equal to 1 since the upper part of the index can fit in primary memory
- **Equality Search Cost:** $C = 1 + 1$
- **Range Search Cost:** in order to explain this we have to introduce additional measures:
  - *Selectivity factor:* $sf(p)$ with $p = v_1 < k < v_2$ is an estimate of the fraction of records which will satisfy the condition:

$$sf(p) = \frac{v_2 - v_1}{k_{max} - k_{min}}$$

  - **Visited leafs:** so how many pairs we can fit on the pages

$$N_{leaf}(idx) = \left\lceil \frac{L_k + L_{RID} \times N_{REC}(R)}{D_{pag} \times f_r} \right\rceil$$

  - Let us assume that the cost of an index access is estimated with the number of leaves to visit $\lceil sf(p) \times N_{leaf}(idx) \rceil$ and the number of RID of the records that satisfy the condition is

$$E_{rec} = \lceil sf(p) \times N_{rec}(R)$$

  - Since the records are not sorted on the index key values, the number of pages to read is E_{rec}, and therefore the search cost using the index is:

$$\lceil sf(p) \times (N_{leaf}(idx) + N_{rec}(R))$$

Clustered Index

- **Search Cost:** $C = CiCd$
- **Equality Search Cost:** $C = 1 + 1$
- **Range Search Cost:** The records are always retrieved using the index since, although it has been constructed from sorted records, in the case of subsequent insertions it is not certain that the records are still sorted. The number of data pages to visit to find E_{rec} records is:

$$\lceil sf(p) \times N_{pag}(R) \rceil$$

  And the overall cost of the search with the use of the index becomes:

$$\lceil sf(p) \times (N_{leaf}(idx) + N_{pag}(R)) \rceil$$

Chapter 19

Non-Key Attribute Organizations

Other important organizations are the ones to retrieve records of a table that satisfy a query which involves nonkey attributes, i.e. attributes that do not uniquely identify a record.

19.1 Non-Key Attribute Search

The records to retrieve are specified with search conditions of the following types:

- An *equality search* which specifies a value v for an attribute $Ai(Ai = vi)$
- A *range search*, which specifies a range of values for an attribute $A_i(v_1 \leq A_i \leq v_2)$
- A *boolean search*, which consists of the previous search types combined with the operators AND, OR and NOT

Sales

RID	Date	Product	City	Quantity
1	20090102	P1	Lucca	2
2	20090102	P2	Carrara	8
3	20090103	P3	Firenze	5
4	20090103	P1	Arezzo	10
5	20090103	P1	Pisa	1
6	20090103	P4	Pisa	8
7	20090103	P2	Massa	5
8	20090104	P2	Massa	2
9	20090105	P4	Massa	2
10	20090103	P4	Livorno	5

Index

Quantity	RID
1	5
2	1
2	8
2	9
5	3
5	7
5	10
8	2
8	6
10	4

Figure 19.1: An Index on Quantity

New have the following proprieties:

- With a primary organization, queries on non-key attributes can only be answered with a scan of the data. With large collections of records and queries satisfied by small subsets, if the response time is an important requirement, this approach is not worthwhile.
- Indexes are non-exclusive. Therefore they can be created for any non-key attributes, regardless of the primary organization used to store the table records.

- An excessive number of indexes can be harmful to the overall performance
- A typical implementation of an index is the inverted index organization

19.2 Inverted Indexes

An inverted index I on a non-key attribute K of a table R is a sorted collection of entries of the form $(k_i, n, p_1, p_2, ..., p_n)$ where:

- k_i is a specific value if K
- n is the number of records containing that value
- $p_1, p_2, ..., p_n$ is the *sorted* RID list of these records

Sales

RID	Date	Product	City	Quantity
1	20090102	P1	Lucca	2
2	20090102	P2	Carrara	8
3	20090103	P3	Firenze	5
4	20090103	P1	Arezzo	10
5	20090103	P1	Pisa	1
6	20090103	P4	Pisa	8
7	20090103	P2	Massa	5
8	20090104	P2	Massa	2
9	20090105	P4	Massa	2
10	20090103	P4	Livorno	5

Inverted index

Quantity	n	RID list
1	1	5
2	3	1, 8, 9
5	3	3, 7, 10
8	2	2, 6
10	1	4

Figure 19.2: An *inverted index* on Quantity

The advantage of using an **inverted index** are the following:

- The data file is accessed only to find the records that match the query
- The result of "count" with conditions, or selection of index attributes can be satisfied with the use of only the index
- The organization of the file is independent from the organization of the index

19.2.1 Performance Evaluation - (NOT REQUIRED)

The performance is evaluated in terms of:

- The amount of extra memory needed, not counting the memory needed to store the data file
- Cost of the search for records with specified values for the indexed attributes, and of the update operations

Memory requirements

$$M = \text{Index memory}$$
$$= \sum_{i=1}^{N_i(R)} N_{key}(I_i)(L_{I_i} + L_R) + N_I(R) \times N_{rec}(R) \times L_{RID}$$
$$\approx N_I(R) \times N_{rec}(R) \times L_{RID}$$

19.2.2 Equality Search - (NOT REQUIRED)

$$C_s = C_I + C_D$$

Where:

- C_s is the cost of accessing the index pages
- C_D is the cost of accessing the data pages containing the records

The **selectivity factor** is determined by the multiplicity of keys

$$sf(cond) = sf(A_i = v_i) = \frac{1}{N_{key}(A_i)}$$

C_I is usually approximated to the cost of accessing the leaf nodes, ignoring the cost of the visit of the path from the root to a leaf node

$$C_I = \lceil sf(p) \times N_{leaf}(I) \rceil = \left\lceil \frac{N_{leaf}(I)}{N_{key}(I)} \right\rceil$$

Unclustered Index

If the index is **unclustered** it is necessary to have an estimate of the number of records E_{rec} satisfying the query condition $(A_i = v_i)$

$$E_{rec} = \lceil sf(p) \times N_{rec}(R) \rceil = \left\lceil \frac{N_{rec}(R)}{N_{key}(I)} \right\rceil$$

The following formula is used to estimate C_D:

$$C_D = \lceil \Phi(E_{rec}, N_{pag}(R) \rceil$$

Where the $\Phi(k, n)$ is an estimate of the number of pages, in a file of n pages, that contain at least one of the k records to be retrieved using a *sorted* rid-list. The following approximation is proposed by Cardenas:

$$\Phi(k, n) = n\left(1 - \left(1 - \frac{1}{n}\right)^k\right)$$

$$\Phi(k, n) \leq \min(k, n)$$

Clustered Index

If the index is **clustered**, otherwise we have:

$$C_D = \lceil sf(p) \times N_{pag}(R) \rceil = \left\lceil \frac{N_{pag}(R)}{N_{key}(I)} \right\rceil$$

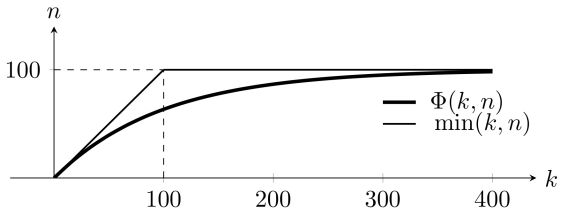


Figure 19.3: Shape of Φ function

19.2.3 Range Search - (NOT REQUIRED)

Let us assume that to retrieve the records that satisfy the condition $v_1 \leq A_i \leq v_2$, an equality search operation is performed for each Ai value in the range. The cost is:

$$C_s = C_I + C_D$$

Where

- $C_I = \lceil sf(p) \times N_{leaf}(I) \rceil$
- $$sf(p) = \frac{v_2 - v_1}{\max(A_i) - \min(A_i)} \quad v_1 \leq A_i \leq v_2$$
- $C_D = NoIndexKeyValues \times NoPageAccessesForRidList$
- $NoIndexKeyValues = \lceil sf(p) \times N_{key}(I) \rceil$
- $NoPageAccessesForRidList$ depends on the index type:
  - **Clustered**
    $$C_D = \lceil sf(p) \times N_{pag}(R) \rceil = \left\lceil \frac{N_{pag}(R)}{N_{key}(I)} \right\rceil$$
  - **Unclustered with ordered lists of RID**
    $$C_D = \left\lceil \Phi\left(\frac{N_{rec}(R)}{N_{key}(A_i)}, N_{pag}(R) \right) \right\rceil$$

19.3 Bitmap indexes

A bitmap index I on a non-key attribute K of a table R, with N records, is a sorted collection of entries of the form (k_i, B), where each values k_i of K is followed by a sequence of N bits, where the j-th bit is set to 1 if the record j-th has the value k_i for the attribute K. All other bits of the bitmap B are set to 0.

The advantages of Bitmap Index are the following one:

- Useful when the attribute has a low selectivity
- Efficient bit operations for set-oriented operators
- Optimal for queries which count the data

Sales

RID	...	Quantity
1	...	2
2	...	8
3	...	5
4	...	10
5	...	1
6	...	8
7	...	5
8	...	2
9	...	2
10	...	5

Bitmap index

1	2	5	8	10
0	1	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
1	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	0	0	0
0	0	1	0	0

Figure 19.4: A *bitmap index* on Quantity

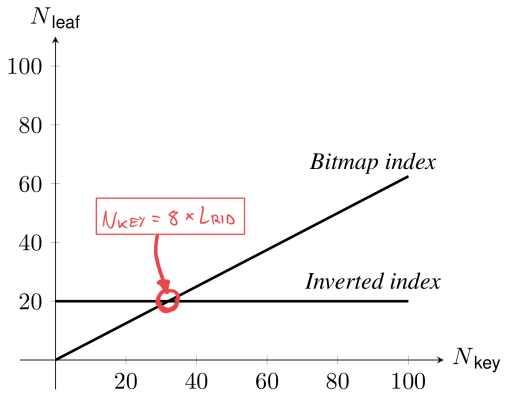
19.3.1 Comparison with traditional indexes - (NOT REQUIRED)

The number of **leaves** of an **inverted index** is:

$$N_{leaf} = \frac{(N_{key}) \times L_k + N_{rec} \times L_{RID}}{D_{pag}} \approx \frac{N_{rec} \times L_{RID}}{D_{pag}}$$

The number of **leaves** of a **bitmap index** is:

$$N_{leaf} = \frac{(N_{key}) \times L_k + N_{key} \times \frac{N_{rec}}{8}}{D_{pag}} \approx N_{key} \times \frac{N_{rec}}{D_{pag} \times 8}$$



19.4 Multi-attribute Index

Let us see how to use inverted indexes to speed up the search of records that satisfy a conjunction of equality or range conditions on a subset of k attributes $A_1, A_2, ..., A_k$.

A query with a condition that uses all the k attributes is called *exact match query*, otherwise is called *partial match query*.

Let $R(A_1, A_2, A_3)$ be a relation with $N_{rec}(R)$ records. The attributes A_1, A_2, A_3 have n_1, n_2 and n_3 distinct values. Three inverted indexes on each attribute have $(n_1 + n_2 + n_3)$ elements, and the total number of RIDs is $3 \times N_{rec}(R)$.

An alternative solution to speed up the search consists in building a **multi-attribute (composite) index** on A_1, A_2, A_3, with $(n1 \times n2 \times n3)$ elements, one for

each combination of the values of the attributes A_i, with the rid-lists of the records that have those three values in the three attributes. The total number of RIDs is now $N_{rec}(R)$.

Chapter 20

Multidimensional Data Organizations

Let us consider multidimensional data representing points or regions in a k-dimensional space. For example, the k attributes of the records of a relation can be interpreted as coordinates of a k-dimensional space and a table of N_{rec} record as N_{rec} spatial points.

Consider a set of 8 records with two attributes A_1 as latitude and A_2 as longitude of cities, both normalized in the range 0 to 100.

City	A_1	A_2
C1	10	20
C2	80	30
C3	40	40
C4	10	85
C5	20	85
C6	40	80
C7	20	55
C8	20	65

Figure 20.1: Multidimensional representation of data on cities

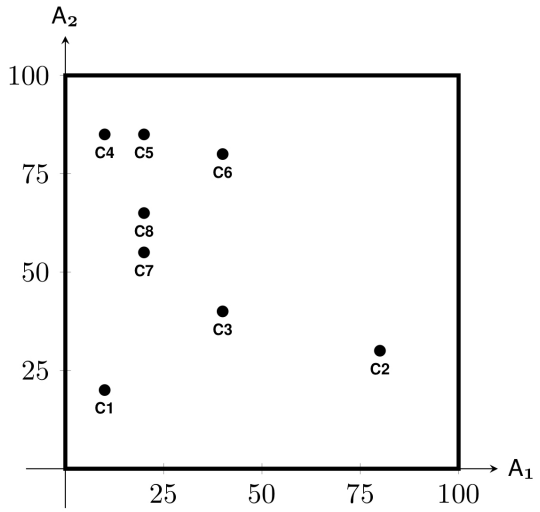


Figure 20.2: Data on cities locations

The problems that will be considered are:

1. **(Primary organization)** How to divide stored data across pages?
2. **(Secondary organization)** How to divide index entries across index levels?

The two questions depends on the type of queries supported:

- *Point / region search:* check if point/region is present
- *Spatial range search:* points/regions in a (hyper)rectangle or (hyper)ball

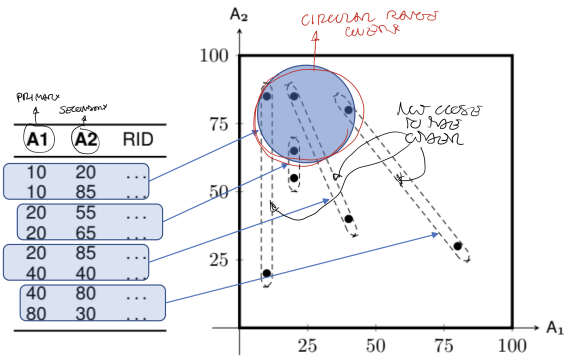
- *k-Nearest neighbors (k-NN):* k points/regions that are the closest ones with respect to a selected point/region

We have multiple choice in order to split our data in a multidimensional organizations, like:

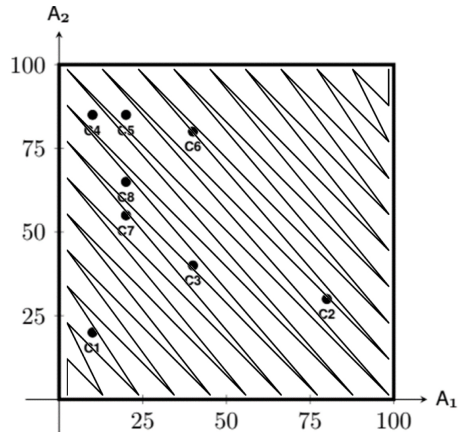
- **Linear** order based organizations, traditional indexes to map the position
  - How to build a total order on multidimensional data?
- Space **partitioning** organizations, referred to the hash approach
  - How to partition the space in regions that contain records that can be stored in a page?
  - How to quickly find the region containing the points in a specified rectangular area?
- **Generalized search tree** based organizations, forget about the space and it focus on the data, it divide the point s to build something that is easy to search
  - How to choose (overlapping) regions that contain record to be stored in a page?
  - How to quickly find region(s) containing the points for a specific query?

20.1 Linearization- B$^+$-tree

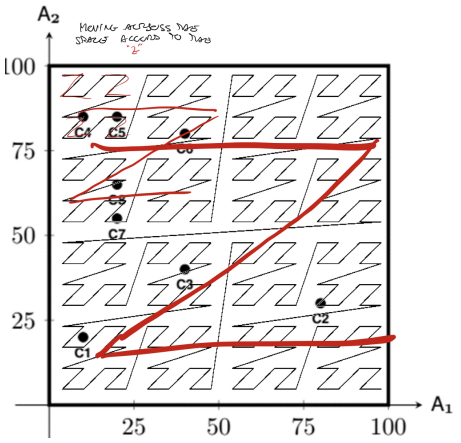
- **Multi-attribute lexicographic order**
  - The dotted lines indicate the linear order in which points are stored in a B+–tree
  - The boxes how points are stored in a multidimensional index
  - Good for point search
  - Almost useless for range search and k-NN. Because depending on the key we are not even able to find a single interesting page



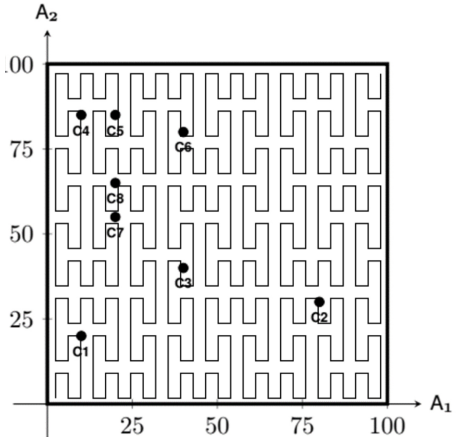
- **Diagonal order (sum of coordinates)**



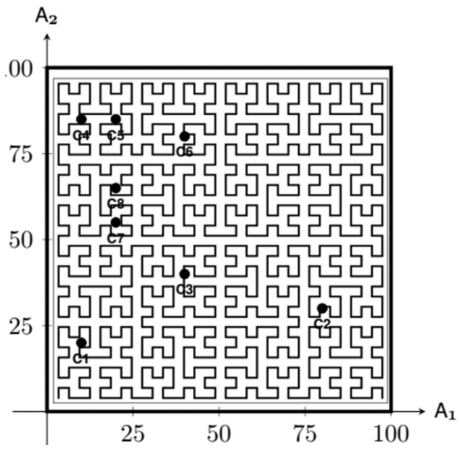
- **Morton space filling curve, Z-order**



- **Peano space filling curve**

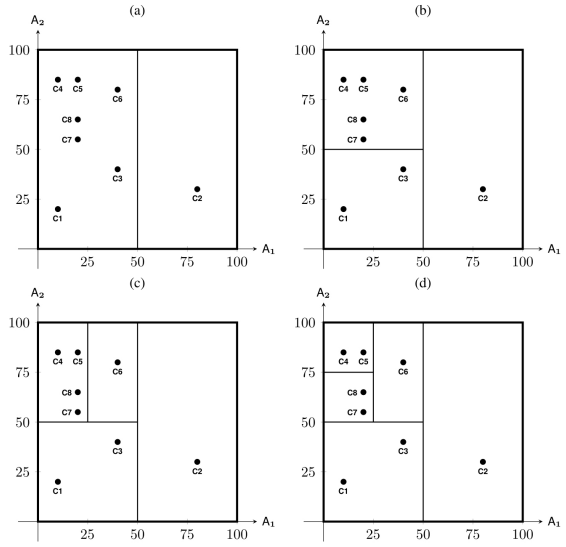


- **Hilbert space filling curve**

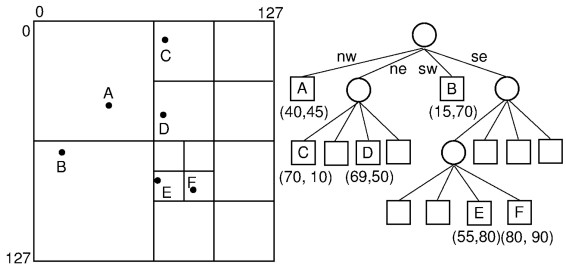


20.1.1 Space partitioning

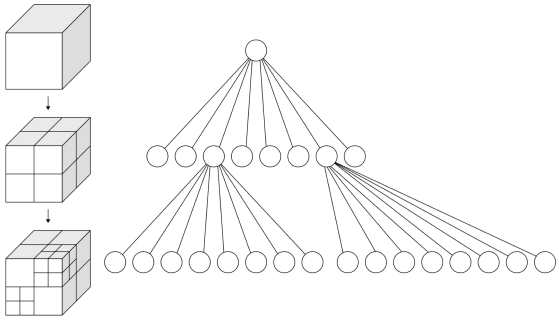
- **Classical**
  - Suppose that pages have a capacity 2 and we want to load the data on cities. After the insertion of "C1" and "C2", the insertion of "C3" requires distributing the data into two pages.
  - To proceed let us divide the data space into non-overlapping partitions according to the first coordinate by choosing a value of separation d for A_1: points with coordinate $A_1 \leq d$ are inserted into a page, those with a higher value are inserted into another one.
  - When there is a new overflow from a page during data loading, we proceed with another split of the partition, but changing the reference coordinate, with the logic that the splitting dimension alternates.



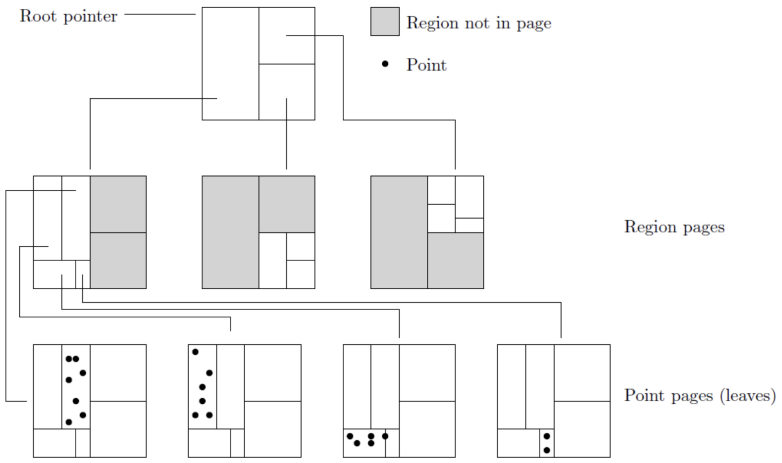
- **Point region quadtrees**
  - Used to organize bidimensional data, evolution of the binary tree to 2D
  - 2 thresholds splits the domain in 4 parts, reason of the name



- **Octrees**
  - Create regions that are not new
  - Higher is the dimension, higher will be the number of nodes that we have to generate
  - Splitting in the middle could be not the best choice



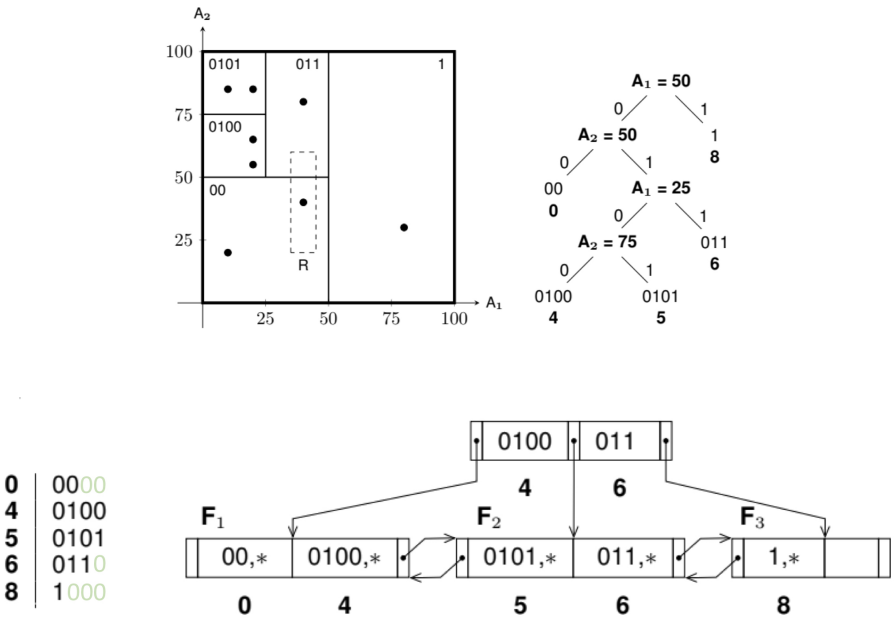
- **KDB-trees**
  - Inherit the idea to have a threshold to split the domain, but it uses a coordinate at time and we circle the order of splitting strategy
  - Every part is splitted when needed
  - How do we find which is the right position?
    * Start from the root is easy
    * Start from bottom → additional tree to take care on the partition volume: **G-trees**



20.2 G-trees

The data space is divided into non-overlapping partitions of variable size identified by an appropriate **code**, then a total ordering is defined for partition codes, and they are stored in a B+–tree. A partition code is a binary string constructed as follows:

- The initial region is identified by the empty string
- Splitting along side X, the two partition are identified by "0" and "1".
  - The points $0 < x \leq 50$ belong to partition "0"
  - The points $50 < x \leq 100$ belong to partition "1"
- In general a partition R with the code S is split, the sub-partition with values **less** than the half interval has the code $S\backslash 0$" and that with values **greater** has the code $S\backslash 1$".



Some important properties of the partition coding:

- If a partition R has the code S, the code of the partition from which R has been created with one split is S without the last bit
- The length of the code of a partition R is the number of split made to get R from the initial space
- Let *RegionOf(S)* be a function that maps the code S of a partition R in the coordinates of the lower left and upper right vertices of the partition

20.2.1 Point Search

The search of a point P with coordinates (x, y) proceeds as follows:

1. The partition tree is searched for the code S_P of the partition that contains P, if it is present.
2. The G–tree is searched for the partition code S_P to check if P is in the associated page

20.2.2 Spatial Range Search

A spatial range search looks for the points P_i with coordinates (x_i, y_i) such that $x_1 \leq x_i \leq x_2$ and $y_1 \leq y_i \leq y_2$. The query result is found as follows:

1. The G–tree is searched for the leaf node F_h of the partition containing the **lower left** vertex (x_1, y_1) of R
2. The G–tree is searched for the leaf node F_k of the partition containing the **upper right** vertex (x_2, y_2) of R
3. For each leaf from F_h to F_k the element's code S are searched such that $R_S =$ *RegionOf(S)* overlaps with the query region R

20.2.3 Point Insertion

The insertion of a point P with coordinates (x, y) proceeds as follows:

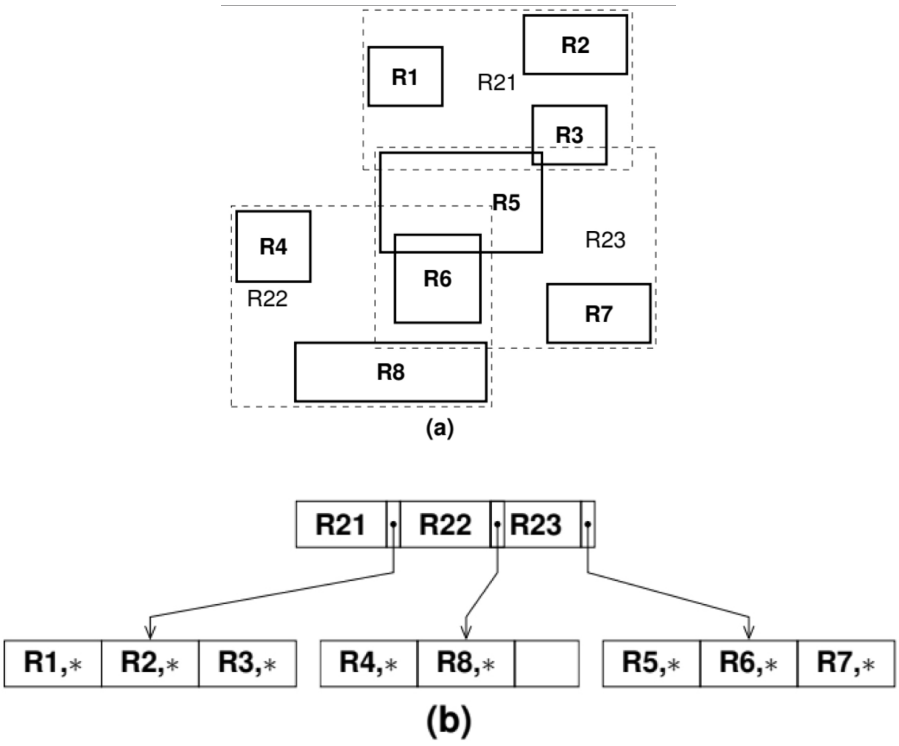
1. The G–tree is searched for the leaf node F of the partition R_P that should contain it. Let S_P be the code of R_P
2. If R_P is not full, insert P, otherwise R_P is split in R_{P_1} and R_{P_2}, with codes $S_{P_1} = S_P$"0" and $S_{P_2} = S_P$"1". If the new strings have a length greater than M, M takes the value $M + 1$
3. The points in R_P and P are distributed in R_{P_1} and R_{P_2}
4. The element (S_P, p_{R_P}) in the leaf F is replaced by $(S_{P_1}, p_{R_{P_1}})$ and $(S_{P_2}, p_{R_{P_2}})$

20.2.4 Point Deletion

The deletion of a point P proceeds as follows:

1. Let F be the leaf node with the partition R_P containing P, S_P the partition code of S_P, and S' the partition code of R' obtained from R_P with a split and therefore different from S_P for the last bit only
2. P is deleted from R_P and then two cases are considered:
   (a) *R' has been split:*
      - If the partition R_P becomes empty, then S_P is deleted
      - Otherwise the operation terminates
   (b) *R' has not been split:*
      - If the two partition cannot be merged, the operation terminates
      - Otherwise the two partition are merged

20.3 R*-trees

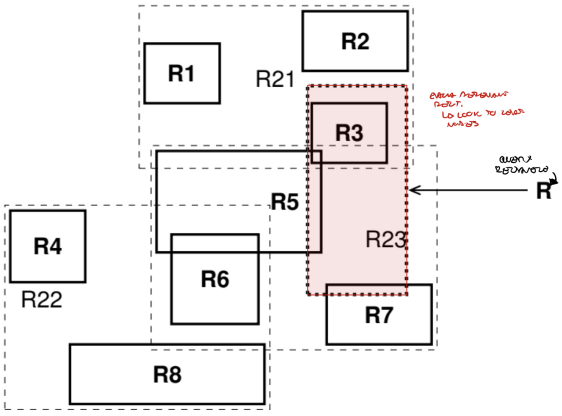


- Variant of R-tree, is a s a dynamic tree structure perfectly balanced as a B+–tree
- Rectangles described by the coordinates of the bottom left and the top right vertices
- Terminal nodes of a R*–tree contain elements of the form (R_i, O_i)
  - R_i is the rectangular data
  - O_i is a reference to the data nodes
- The non-terminal nodes contain elements of type (R_i, p_i)
  - p_i is a reference to the root of a subtree
  - R_i is the minimum bounding rectangle containing all rectangles associated with the child nodes

20.3.1 Search Overlapping Data Regions

The search of the data regions overlapping with R proceeds as follows:

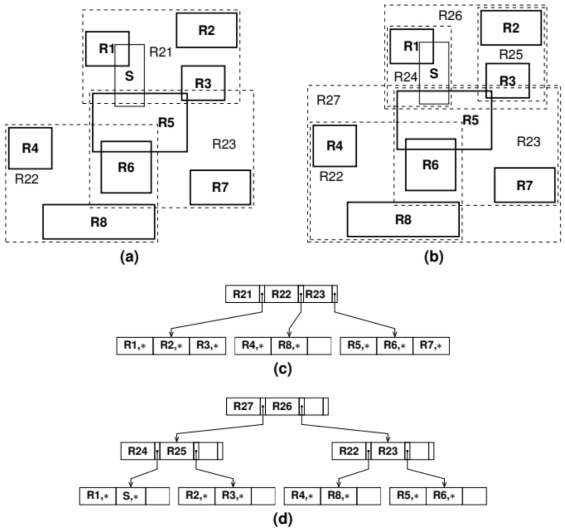
- The root is visited in order to look for the elements (R_i, p_i) with R_i that overlaps with R
- For each element (R_i, p_i) found, the search proceeds in a similar way in the subtrees rooted at p_i
- When a leaf node is reached, the data regions R_i in the search result are those with R_i that overlaps with R



20.3.2 Insertion

Let S be a new data region to insert in the tree. The operation is similar to inserting a key in a B+–tree since S is stored into a leaf node, and if there is an overflow, the node will be split into two nodes. In the worst case the division can propagate to the parent node up to the root. The choice of the region in an internal node can be made according to the degree of overlap with S. After having selected the leaf node N where to insert S, if an overflow does not occur, the region is recalculated and its value propagates in the parent node. Otherwise, two cases are considered:

- **Forced Reinsert:** If this is the first overflow from a leaf node, it is not split instead p of the $(M + 1)$ entries are removed from the node and reinserted in the tree
- **Node Splitting:** After the first overflow, the (M + 1) elements are divided between two nodes, and two new elements are inserted in the parent node, and the process goes on to propagate the effects



Chapter 21

Access Methods Management

The *Access Methods Manager* provides to the *Relational Engine* the operators used by its modules to execute the commands for the definition and use of databases.

21.1 The Storage Engine

An abstract database machine is normally divided into two parts:

- An abstract machine for the logical data model, **Relational Engine**
- An abstract machine for the physical data model, **Storage Engine**

The Relational Engine includes modules to support the execution of SQL commands, by interacting with the Storage Engine.

While the interface of the relational engine depends on the data model features, the interface of the storage engine depends on the data structures used in permanent memory.

The operators on data exported by the storage engine are procedural and can be grouped into the following categories:

- Operators to *create* databases
- Operators to *start* and to *end* a transaction
  - *beginTransaction*
  - *commit*
  - *abort*
- Operators on *heap files* and *indexes*
- Operators about *access methods* available for each relation

21.2 Access Method Operators

The Access Methods Manager provides the operators to transfer data between permanent memory and main memory in order to answer a query on a database.

Permanent data are organized as collections of records, stored in **heap files**, and indexes are optional auxiliary data structures associated with a collection of records.

An **index** consists of a collection of records of the form *(key value RID)*, where

- *key value* is a value for the search key of the index
- *RID* is the identifier of a record in the relation being indexed

The operators provided by the Access Methods Manager are used to implement the operators of physical query plans generated by the query optimizer.

The records of a heap file can be retrieved by a serial scan or directly using their RID obtained by an index scan with a search condition.

A heap file or index scan operation is implemented as an iterator, also called cursor, which is an object with methods that allow a consumer of the operation to get the result one record at a time.

When a heap file iterator is created, it is possible to specify the RID of the first record to return. Instead, when an index iterator is created a key range is specified.

Chapter 22

Implementation of Relational Operator

22.1 Assumption and Notation

22.1.1 Physical Data Organization

- Let us assume that each relation has attributes *without null value*s, and it is stored in a heap file, or with the primary organization *index sequential*.
- The indexes are organized as a B+–tree and those for non-key attributes are inverted indexes
- We also distinguish two types of indexes:
  - A **clustered index** is built on one or more attributes of a relation sorted on the index key
  - An **unclustered index** is built on one or more attributes which are not used to sort a relation

22.1.2 Physical Query Plan Operators

The query optimizer has the task of determining how to execute a query in an "optimal" way, by considering the physical parameters involved. Each relational algebra operator can be implemented in different ways and the query optimizer must use appropriate strategies to estimate the costs of the alternatives and choose the one with lowest cost.

22.1.3 Physical Query Plan Execution

Physical operators are implemented as *iterators* that produce the records of the result one at a time on request. An iterator behaves as an *object with state*, has a certain *type*, can return the records in a certain order and has the following methods:

- **open:** Initializes the process of getting records
- **isDone:** Tests if the iterator has more data to return

- **next:** Returns the next result record
- **reset:** Re-initializes the iterator already open
- **close:** Performs some clean-up operations and ends the iterator

Physical operators are either *blocking* or *non-blocking*. A blocking operator, when is opened, must call next exhaustively on its operands before returning its first (or next) record (e.g. the sort operator)

22.2 Physical Operators for Relation R

The records of a relation can be retrieved with any of the following operators, which have a relation R as an argument.

- **TableScan(R)** The operator returns the records of R, in the order that they are stored
$$C = N_{pag}(R)$$
- **SortScan($R, \{A_i\}$)**
  - The operator returns the records of R sorted in ascending order on the set of attributes Ai values
  - Sorting is done with the **merge sort** algorithm
  - The operator's cost depends on the $N_{pag}(R)$ value, the number of pages B available in the buffer and the implementation of the *merge sort*
  - If the merge sort is implemented so that it returns the final result of the merge without first writing it to a temporary file, the cost will be more or less:
$$N_{pag}(R) = B \times (B - 1)$$
- **IndexScan(R, I)**
  - The operator returns the records of R sorted in ascending order on the set of attributes $\{A_i\}$
  - Cost that depends on the type of index and the type of attribute on which it is defined.
$$C = \begin{cases} N_{leaf}(I) + N_{pag}(R) & \text{if } I \text{ is clustered} \\ N_{leaf}(I) + N_{rec}(R) & \text{if } I \text{ is unclustered} \\ N_{leaf}(I) + \lceil N_{key}(I) \times \Phi(\lceil N_{rec}(R)/N_{key}(I) \rceil, N_{pag}(R)) \rceil & \text{if } I \text{ is an inverted index} \end{cases}$$
- **IndexSequentialScan(R,I)**
  - The operator returns the records of R, stored with the primary organization *index sequential* I
  - Sorted in ascending order on the primary key values
$$C = N_{leaf}(I)$$

22.3 Physical Operator for Projection π^b

The physical operator that implements **projection**, without duplicate elimination, has as first argument the records returned by O, another physical operator.

- **Project**$(O, \{A_I\})$
  - The operator returns the records of the projection of the records of O over the set of attributes $\{A_i\}$
  - No cost a part of the computational cost that can be ignored
$$C = C(O)$$
- **IndexOnlyScan**$(R, I, \{A_i\})$
  - Return the sorted record over the projection to only a specific set of attributes using sorted index I of the relative attributes $\{A_i\}$
  - Usage of index since they are less heavy than records
$$C = N_{leaf}(I)$$

22.4 Physical Operators for Duplicate Elimination δ

The physical operators for **duplicate elimination** have as argument the records returned by O, another physical operator.

- **Distinc**(O)
  - The records of O must be *sorted* so that duplicates will be next to each other
  - Returns the records of O sorted, without duplicates
$$C = C(O)$$
- **HashDistinc**(O)
  - Returns the records of O without duplicates using a hash technique
  - We have $B + 1$ pages in the buffer to perform the duplicate elimination. Composed in two phases each one has a specific hash function
    * h_1 - *partitioning phase:* for each record of O the hash function h_1 is used to distribute all the records uniformly in the B pages. When a page is full it is written in the respective partition file. At the end we have a partition of the records of O in B files, each containing a collection of records that share a common hash value
    * h_2 - *duplicate elimination phase:* each partition is read page-by-page to eliminate duplicates with the hash function h_2 applied to all record attributes. A record r is discarded only when it collides with another record r' with respect h_2
$$C = C(O) + 2 \times N_{pag}(O)$$
  - Same cost of distinct with the sorting of the operand records
  - Disadvantage of not producing a sorted result

22.5 Physical Operators for Selection σ

- **Filter**(O, ψ)
  - Returns the records of O satisfying the condition ψ without indexes

$$C = C(O)$$

- **IndexFilter**(R, I, ψ)
  - Returns the records of R satisfying the condition ψ with the use of the index I
  - The result will be sorted
  - The operator is composed on two phases:
    - It uses the index to find the sorted set of RIDs of the records satisfying the condition: RIDIndexFilter(I, ψ)
    - Retrieves the records of R: TableAccess(O, R)

$$C = C_I + C_D$$

- **IndexSequentialFilter**(R, I, ψ)
  - Returns the sorted records of R, stored with an **index sequential I** satisfying the condition ψ

$$C = \lceil s_f(\psi) \times N_{leaf}(I)$$

- **IndexOnlyFilter**$(R, I\{A_i\}, \psi)$
  - Return the sorted records of a selection with projection using only the index I

$$C = \lceil s_f(\psi) \times N_{leaf}(I)$$

- **OrIndexFilter**$(R, \{I_i, \psi_i\})$
  - returns the records of R satisfying the *disjunctive* condition: $\psi = \psi_1 \vee \psi_2 \vee ... \vee \psi_n$ using all the indexes I_i one for each term ψ_i
    1. Uses the indexes to find a sorted union of the RID list matching each terms ψ_i
    2. Return the records of R

$$C_I = \left\lceil \sum_{k=1}^{n} C_I^k \right\rceil$$

$$E_{rec} = \lceil s_f(\psi) \times N_{rec}(R)$$

$$C_D = \lceil \Phi(E_{rec}, N_{pag(R)}) \rceil$$

- **AndIndexFilter**$(R, \{I_i, \psi_i\})$
  - returns the records of R satisfying the *conjunctive* condition: $\psi = \psi_1 \wedge \psi_2 \wedge ... \wedge \psi_n$ using all the indexes I_i one for each term ψ_i

1. Uses the indexes to find a sorted intersection of the RID list matching each terms ψ_i
2. Return the records of R

$$C_I = \left\lceil \sum_{k=1}^{n} C_I^k \right\rceil$$
$$E_{rec} = \lceil s_f(\psi) \times N_{rec}(R)$$
$$C_D = \lceil \Phi(E_{rec}, N_{pag(R)}) \rceil$$

22.6 Physical Operators for Grouping γ

The records of R are partitioned according to their values in one set of attributes $\{A_i\}$ called *grouping attributes*. Then, for each group, the values in certain other attributes are aggregated with the functions $\{f_i\}$.

- **GroupBy**$(O, \{A_i\}, \{f_i\})$
  - To group the records of O on the set of attributes $\{A_i\}$ using the aggregation function in $\{f_i\}$
  - In the set $\{f_i\}$ there are the aggregation function present in the SELECT and HAVING clauses
  - The records of O are sorted on the $\{A_i\}$

$$C = C(O)$$

- **HashGroupBy**$(O, \{A_i\}, \{f_i\})$
  - The records of O are partitioned using a hash function on the attributes $\{A_i\}$
  - Two phases:
    1. *partitioning phase:* a partition is created using the hash function h_1
    2. *grouping phase:* the records of each partition are grouped with the hash function h_2

$$C = C(O) + 2 \times N_{pag}(O)$$

22.7 Physical Operators for Join $\bowtie$

The cartesian product plus the filter create all possible tuple and remove all pairs that doesn't match the foreign key, but this is not efficient.

We will consider the equijion computed with one of the following physical operators, where:

- O_E is the external operand
- O_I is the internal operand
- ψ_I is the join condition

```
for each r ∈ O_E do
    for each s ∈ O_I do
        if ψ_J then add < r, s > to the result;
```

- **NestedLoop**(O_E, O_I, ψ_J)
  - It verify if every possible pair satisfy the predicate ψ

$$C_{NL} = C(O_E) + E_{rec(O_E) \times C(O_I)}$$

- **PageNestedLoop**(O_E, O_I, ψ_J)
  - The cost of the nested loop join can be reduced considerably if, instead of scanning the result of O_I once per record of O_E, we scan it once per page of O_E
  - The inner relation is scanned only once for page of the puter relation

$$C_{PNL} = C(O_E) + N_{pag}(O_E) \times C(O_I)$$
$$C_{PNL} = N_{pag}(S) + N_{pag}(S) \times N_{pag}(R)$$

  - This method is better than the NestedLoop since $N_{pag}(R) < N_{rec}(R)$, but it has the defect of producing an unsorted result

```
for each page p_r of O_E do
    for each page p_s of O_I do
        for each r ∈ p_r do
            for each s ∈ p_s do
                if ψ_J then add < r, s > to the result;
```

- **IndexNestedLoop**(O_E, O_I, ψ_J)
  - Only with *equi-join*
  - In case there is an index on the join column of one relation, we can exploit it
  - Still goes through the external operator
  - Use the index to restrict the search, no need to check the condition since it is already satisfied

```
for each r ∈ O_E do
    for each s ∈ IndexFilter(S, I, S.A_j = r.A_i) do
        add < r, s > to the result;
```

$$C_{INL} = C(O_E) + E_{rec}(O_E) \times (C_I + C_D)$$

  - $(C_I + C_D)$ is the cost to retrieve the matching records of S with a record of O_E that depends on the index type

- **MegeJoin**(O_E, O_I, ψ_J)
  - Only with *equi-join* plus sort-merge

  - O_E and O_I records are *sorted* on the join attributes columns
  - The O_E join attribute is a key
  - Join attribute is a key

$$C_{MJ} = C(O_E) + C(O_I)$$

- **HashJoin**(O_E, O_I, ψ_J)
  - In case we do not have sorted data we can use two distinct hash functions in two phases in order to compute the join
    1. *partitioning phase:* the records of O_E and O_I are partitioned using the function h_1

$$C(O_E) + C(O_I) + N_{pag}(O_E) + N_{pag}(O_I)$$

    2. *probing phase:*
       * For each partition the records of O_E are read and inserted into the buffer hash table with B pages using the function B
       * The records of O_I are read one page at a time and the one joining with O_E are checked with h_2
       * And added to the result

$$N_{pag}(O_E) + N_{pag}(O_I)$$

  - The total cost is:

$$C_{HJ} = C(O_E) + C(O_I) + 2 \times (N_{pag}(O_E) + N_{pag}(O_I))$$

22.8 Physical Operators for Set and Multiset Union, Intersection and Difference

Union(O_E, O_I)**, Except**(O_E, O_I)**, Intersect**(O_E, O_I):

- Require that the records of the operands are sorted and without duplicates
- The variants with **All** do note require to eliminate the duplicates
- The **UnionAll** is simple and powerful
- They all have cost equal to:

$$C = C(O_E) + C(O_I)$$

Chapter 23

Query Optimization

- **Goal of optimization:** to find an efficient plan to execute a query
- **Access plan:** tree of physical operator
- **Query optimization:** important task in a DBMS, and it is divided into 4 phases:
  1. *Query analysis:* the correctness of the SQL query is checked
  2. *Query transformation:* where the initial logical plan is transformed into an equivalent one that provides a better query performance
  3. *Physical plan generation:* alternative algorithms for the query execution are considered
  4. *Query evaluation:* the physical plan is executed

23.1 Query Analysis Phase

1. Lexical and syntactic query analysis. Like type checking and user's access rights
2. Simplification of conditions (where) with:
   - The equivalence rules of boolean expression
   - Conversion into conjunctive normal form
   - Elimination of contradiction
   - Usage of Dde Morgan rules to eliminate NOT before simple expression
3. Transformation into tree of relational algebraic operators

23.2 Query Transformation Phase

- Selections are pushed below projections
- Selections are grouped
- Selections and projections are pushed below an inner join

- Unnecessary projections are eliminated
- Projections are grouped

23.2.1 DISTINCT Elimination

- DISTINCT require SORT, which is a costly operation
- We use the functional dependency theory in order to discover if an interesting functional dependency can be inferred into the result of a query

23.2.2 GROUP BY Elimination

- GROUP BY require SORT, which is a costly operation
- It can be eliminated in the following two cases:
  - If there is only a single group
  - Each group is composed by a single tuple

23.2.3 WHERE-Subquery Elimination

Subqueries can occur in the WHERE clause through operators =, <, >, ...; through the quantifiers ANY, or ALL; or through the operators EXISTS and IN, and their negations. All these cases can be rewritten using EXISTS and NOT EXISTS. In which in a certain cases can be eliminated with the introduction of JOIN.

View Merging

- Complex queries are much easier to write and understand if views are used
- Instead, the use of the WITH clause provides a way of defining temporary views available only to the query in which the clause occurs
- the optimizer generates a physical sub-plan for the SELECT that defines the view, and optimizes the query considering the scan as the only access method available for the result of the view

23.3 Physical Plan Generation Phase

The goal is to find a plan to execute a query, among the possible ones, which has the minimum cost on the basis of the available information.

The two main steps are:

- Generation of alternative physical plans
- Estimation of the costs and the choice of the lowest cost plan. To estimate the cost of a physical query plan it is necessary to estimate, for each node in the physical tree, the following parameters:
  - The cost of the physical operator
  - The size of the result and if the result is sorted

23.3.1 Single-Relation Queries

If the query uses just one relation, the operations involved are the projection and selection, like:

SELECT bs
FROM S
WHERE FkR > 100 AND cS = 2000

If there are no useful indexes, the solution is to perform a **Scan + Projection**. Otherwise if there are useful index we can:

- **If there are no useful indexes:** If there are different indexes usable for one of the simple conditions, the one of minimal cost is used. Test if the record satisfy the conditions and projection is applied
- **Use of Multiple-Indexes:** the operation is completed by testing whether the retrieved records satisfy the remaining conditions and the projection is applied
- **Use of Index-Only:** If all the attributes of the condition of the SELECT are included in the prefix of the key of an index, the query can be evaluated using only the index with the query plan of minimum cost

Chapter 24

Transaction And Recovery Management

24.1 Transaction

A DBMS provides a programming abstraction called a **transaction**, which groups together a set of instructions that read and write data. And it has the following properties:

- **Atomicity** only transactions terminated normally change the database, otherwise the database remains unchanged
- **Isolation** when a transaction is executed concurrently with others, the final effect must be the same as if it was executed alone
- **Durability** commitment is an irrevocable act

The acronym ACID is frequently used to refer to the following four properties of transactions: *Atomicity*, *Consistency*, *Isolation*, and *Durability*. **Consistency** cannot be ensured by the system when the integrity constraints are not declared in the schema. However, assuming that each transaction program maintains the consistency of the database, the concurrent execution of the transactions by a DBMS also maintain consistency due to the isolation property.

- *Isolation* when a action is executed concurrently with others, the final effect must be the same as a serial execution of committed transactions

The DBMS provide modules that guaranteed the previous properties:

- **Atomicity → Transaction Manager**
- **Durability → Recovery Manager**
- **Isolation → Concurrency Manager**

24.1.1 Tansactions from the DBMS's Point of View

DBMS only "sees" the read and write operations on its data. A write operation updates a page in the buffer, but does not cause an immediate transfer of the page to the permanent memory. So in case of failure the content of the buffer is lost, the updates might not have been written to the database.

A transaction for the DBMS is a sequence of read and write operations which start and end with the following *transaction operations*:

- *beginTransaction* beginning of the transaction
- *commit* successful termination of the transaction, the system has to make its updates durable
- *abort* abnormal termination, the system has to undo its updates

The execution of the **commit** operation does not guarantee the successful termination of the transaction, because it is possible that the transaction updates cannot be written to the permanent memory, and therefore it will be aborted,

Then we list all the **Transaction States**:

- *active* immediately after it starts
- *partially committed* when it ends, but no consistent update has been made at this point
- *committed* if it has been processed successfully and all its updates on the database have been made durable
- *failed* if it cannot be committed or it has been interrupted
- *aborted* if it has been interrupted and all its updates on the database have been undone

24.2 Types of Failures

A centralized database can become inconsistent because of the following types of failures:

- **Transaction Failure:** interruption of a transaction which *does not damage* the content of the **buffer** and *permanent memory*
- **System Failure:** system interruption, where the *content of the buffer* is **lost**, but *content of the permanent memory* remains **intact**
- **Media Failure:** interruption of the DBMS in which the *content of the permanent memory* is corrupted or **lost**

24.3 Database System Model

The permanent memory consists of three main components:

1. The database
2. *Log*
3. *DB Backup*

The last two are used by the recovery procedure in the case of failures. The database, log and backup are usually stored on distinct physical devices.

24.4 Data Protection

To protect a database the DBMS maintains some redundant information during normal execution of transactions so that in the case of a failure it can reconstruct the most recent database state before the occurrence of the failure.

Database Backup

DBMSs provide facilities for periodically making a backup copy of the database.

Log

- The **history** of the operations performed on the database from the last backup is stored in the **log**. For each transaction we write in the log when the transaction: *starts*, *commits*, *aborts*, *modifies* a page
- Each log record is identified through the so called **LSN (Log Sequence Number)**, that is assigned in a strictly increasing order
- The LSN can be viewed as the serial number of the position of the first character of the record in the file.
- The exact content of the log depends on the algorithms
- In general a log is stored in a file buffered for efficiency reasons
- The log is not buffered and each record is immediately written to the permanent memory

Undo and Redo Algorithms

A database update changes a page in the buffer, and only after some time the page may be written back to the permanent memory. We say that a recovery algorithm requires

- An *undo* if an update of some uncommitted transaction is stored in the database, so the recovery algorithm must undo the updates
- *Redo* if a transaction is committed before all of its updates are stored in the database, so the recovery algorithm must redo the updates

A failure can happen also during the execution of a recovery procedure, and this requires the restart of the procedure.

Chechpoint

To reduce the work performed by a recovery procedure in the case of system failure, another information is written to the log, the so called **checkpoint (CKP) event**.

We specify only a single type of Checkpoint, the **Commit-consistent checkpoint**:

1. The activation of new transactions is suspended
2. The systems waits for the completion of all active transactions
3. All modified pages present in the buffer are written to the permanent memory and the relevant records are written to the log. Here the permanent memory is forced so that all the transactions terminated before the checkpoint have their updates
4. The CKP record is written to the log
5. A pointer to the CKP record is stored in a special file, called *restart file*.
6. The system allows the activation of new transactions

Not efficient by the first two steps.

The problem of finding the optimal checkpoint interval can be solved with a balance between the cost of the checkpoint and that of the recovery, but is very complex. Thus more efficient checkpoint method is preferred, called **buffer-consistent checkpoint – Version 1**.

- This method does not need the waiting for the termination of active transactions
- Still presents the problem of the buffer flush operation
- Other methods have been studied, like the ARIES algorithm, that avoids
  - The suspension of the activation of new transactions
  - The execution of active transactions
  - The buffer flush operation

24.5 Recovery Algorithms

The Recovery managers for the transactions management differ in the way they combine the *undo* and *redo* algorithms:

- **Undo-Redo**
- **Undo-NoRedo**
- **NoUndo-Redo**
- **NoUndo-NoRedo**

24.5.1 Use of the Undo Algorithm

Depends on the policy used to *write pages* updated by an active transaction to the database: *Deferred update* and *Immediate update*. these two are called *NoUndo-Undo* policies.

Deferred update policy requires that updated pages cannot be written to the database before the transaction has committed

- To avoid the pages updated by a transaction are *"pinned"* in the buffer until the end of the transaction
- *NoUndo* type: it is not necessary to undo its updates since the database has not been changed

Immediate update policy allows that updated pages can be written to the database before the transaction has committed

- To allow this, the page is marked as *"dirty"* and its pin is removed
- *Undo* type: if a transaction or system failure occurs the updates on the database must be undone

If a database page is updated before the transaction has committed, its before-image must have been previously written to the log file in the permanent memory.

24.5.2 Use of the Redo Algorithm

Depends on the policy used to *commit* a transaction. There are *Deferred commit* and *Immediate commit*. These two policies are called *NoRedo-Redo*.

Deferred commit policy requires that all updated pages are written to the database before the commit record has been written to the log

- *NoRedo* type: it is not necessary to redo all the updates on the database in the case of failure
- Buffer manager is forced to transfer to the permanent memory all the pages updated by the transaction before the commit operation

Immediate commit policy allows the commit record to be written to the log before all updated pages have been written to the database

- *Redo* type: it is necessary to redo all the updates in the case of a system failure
- The buffer manager is free to transfer the unpinned pages to the permanent memory

Redo Rule or Commit Rule Before a transaction can commit, the after-images produced by the transaction must have been written to the log file in the permanent memory.

24.5.3 No use of the Undo and Redo Algorithms

- **NoUndo** algorithm requires that all the updates of a transaction must be in the database *after* the transaction has committed
- **NoRedo** algorithm requires that all the updates of a transaction must be in the database *before* the transaction has committed

Therefore, a NoUndo-NoRedo algorithm requires that all the updates of a transaction must be in the database neither before nor after the transaction has committed.

When a transaction updates for the first time a page:

1. A new database page is created, the *current page* with a certain address p, whereas the old page becomes a **shadow page**
2. The *New Page Table* is updated so that the first element contains the physical address p of the current page

When the transaction reaches the commit point, the system should substitute all the shadow pages with an atomic action, otherwise if a failure happen the database would be left in an incorrect state.

This atomic action is implemented by executing the following steps:

1. The pages updated in the buffer are written to the permanent memory
2. The descriptor of the database is updated with an atomic operation

24.6 Recovery Manager Operations

24.6.1 Undo-NoRedo Algorithm

- Redoing a transaction to recover from a system failure is never necessary
- Only the *before-image* must be written to the log
- This algorithm is generally used with pessimistic concurrency control algorithm that rarely aborts transactions

24.6.2 NoUndo-Redo Algorithm

- Redoing a transaction to recover from a system failure is never necessary
- Only the *after-image* must be written to the log

- The pages updated by a transaction are written to the database when the transaction has committed, but after the writing of the commit record to the log
- Optimistic concurrency control algorithm, which aborts transactions in case of conflict

24.6.3 Undo-Redo Algorithm

- Require both undo and redo, and is the most complicated one
- Both the *after-image* and the *before-image* must be written to the log to deal with transaction and system failures
- The recovery algorithm is more costly, but it is preferred by the commercial DBMS, since it privileges the normal execution of transactions

24.7 Recovery from System and Media Failures

In the case of system failures, in order to recover the database, the **restart** operator is invoked to perform the following steps **(warm restart)**:

- Bring the database in its committed state with respect to the execution up to the to the system failure
- Restart the normal system operations

And is described with two phases:

- In the *Rollback phase* the log is read from the end to the beginning
  - To *undo*, if necessary, the updates of the non terminated transactions
  - To find the set of the identifiers of the transactions which are terminated successfully in order to redo their operations
- In the *rollforward phase* the log is read onward from the first record after the checkpoint to redo all the operations of the terminated transaction

In the case of media failure, that is of the loss of the database but not of the log, a **cold restart** is performed through the following steps:

- The most recent database backup is reloaded
- A *rollback phase* is performed on the log
- A *rollforward phase* is performed to update the copy of the database,

Chapter 25

Concurrency Management

The operations of a transaction may be performed between those of others. This can cause interference that leaves the database in an inconsistent state. The *Concurrency Manager* is the system module that ensures the execution of concurrent transactions without interference during database access.

25.1 Introduction

We assume that each transaction is *consistent*, i.e. when it executes in isolation, it is guaranteed to map consistent states of the database to consistent states.

Serial Execution An execution of a set of transactions $T = T_1, ..., T_n$ is *serial* if, for every pair of transactions T_i and T_j, all the operations of T_i are executed before any of the operations of T_j or vice versa.

Serializable Execution An execution of a set of transactions is *serializable* if it has the same effect on the database as some serial execution of the transactions that commit.

The DBMS module that controls the concurrent execution of transactions is called the **Concurrency Manager or Scheduler**.

The **theory of serializability** a mathematical tool that allows us to prove whether a given scheduler is correct. The theory uses a structure called **history (or schedule)** to represent the chronological order in which the operations of a set of concurrent transactions are executed, and defines the properties that a history has to meet to be serializable.

25.2 History

From the point of view of a DBMS a transaction T_i starts with a *begin*, then continues with a partially ordered sequence of read ($ri[x]$) and write ($wi[x]$) operations on the database, and terminates either with an $abort(a_i)$ or a $commit(c_i)$ operation.

To simplify the presentation, we will make the following additional assumptions:

- The database is a fixed set of independent records that can only be read or updated
- A transaction reads and updates a specific record at most once

To treat formally the concurrent execution of a set of transactions $\{T_1, T_2, ..., T_n\}$ the concept of **history** is used.

History. Let $T = T_1, T_2, ..., T_n$ a set of transaction. A **history** H on T is an ordered set of operations such that:

1. The operation of H are those of $T_1, T_2, ..., T_n$
2. H preserves the ordering between the operations belonging to the same transaction

Intuitively, the history H represents the actual or potential execution order of the operations of the transactions $T_1, T_2, ..., T_n$.

25.3 Serializable History

For equivalent histories we could mean that they produce the same effects on the database. Since a DBMS knows nothing of the computations made by a transaction in temporary memory, a *weaker notion of equivalence* which takes into account only the order of operations in *conflict* made on the database is preferred.

Operations in conflict. Two operations of different transactions are in *conflict* if they are on the same data item and at least one of them is a write operation.

The concept of operations in conflict allows to characterize three types of abnormal situations:

- *Dirty Reads (Write-Read Conflict)*
- *Unrepeatable Read (Read-Write Conflict)*
- *Lost Update (Write-Write Conflict)*

Histories c-equivalent. Two histories H_1 and H_2 are *c-equivalent (conflict-equivalent)*, that is equivalent with respect to operations in conflict, if:

- They (H_1 and H_2) are defined on the same set of transactions and have the same operations
- They (H_1 and H_2) have the same order of operations in conflict of transactions terminated normally

It is motivated by the fact that the result of concurrent execution of $T_1, T_2, ..., T_n$ depends only on the order of execution of operations in conflict.

A history can be transformed into a c-equivalent one using the following property:

Commutative Property. Two database operations o_i and o_j commute if, for all initial database states, they:

1. Return the same results
2. Leave the database in the same final state

Conflict Serializability. A history H on the transactions $T = T_1, T_2, ..., T_n$ is *c-serializable* if it is c-equivalent to a serial history on $T_1, T_2, ..., T_n$.

Each c-serializable history is serializable, but there are serializable histories that are not c-serializable.

Serialization Graph

Although it is possible to examine a history H and decide whether or not it is c-serializable using the commutativity and reordering of operations. The other way is to proceed based on the analysis of a particular graph derived from H, called *serialization graph.*

Serialization Graph. Let H a history of committed transactions $T = T_1, T_2, ..., T_n$. The *serialization graph* of H, denoted $SG(H)$, is a directed graph such as:

- A **node** for every committed transaction in H
- **Directed arc** from $T_i \rightarrow T_j (i \neq j)$ if and only if in H some operation p_i in T_i appears before and conflicts with some operation p_j in T_j.

We say that two transactions T_i and T_j conflicts if $Ti \rightarrow Tj$ appears in $SG(H)$.

C-Serializability Theorem. A history H is c-serializable if and only if its serialization graph $GS(H)$ is *acyclic.*

25.4 Serializability with Locking

The c-serializability theorem is used to prove that the scheduling algorithm for the concurrency control used by a scheduler is correct.

25.4.1 Strict Two-Phase Locking

Each data item used by a transaction has a lock associated with it, a *read* or a *write* lock. The protocol follows two rules:

1. If a transaction wants to *read* a data item, it first request a *shared* lock on the data item. If no other transaction holds the lock, then the data item is locked.

If, however, another transaction T_j holds a lock in conflict, then T_i must wait until T_j releases it

2. All locks held by a transaction T_i are released together when T_i commits or aborts

- A Strict 2PL protocol ensures c-serializability.
- *The set of Strict 2PL histories is a proper subset of the c-serializable histories*

25.4.2 Deadlocks

Two-phase locking is simple, but the scheduler needs a strategy to detect deadlocks.

Deadlock Detection

- A strategy to detect deadlocks uses a wait-for graph in which the nodes are active transactions and an arc from T_i to T_j indicates that T_i is waiting for a resource held by T_j
- Then a cycle in the graph indicates that a deadlock has occurred, and one of the transactions of the cycle must abort
- The standard method to decide which transaction to abort is to choose the "youngest" transaction by some metric
- Linear complexity algorithm
- Instead of *wait-for graph* we can use the *timeout* strategy: if a transaction has been waiting too long for a lock, then the scheduler simply presumes that deadlock has occurred

Deadlock Prevention

Use a locking protocol that disallows the possibility of a deadlock occurring as follows. Each transaction T_i receives a timestamp $t_s(T_i)$ when it starts: T_i is older than T_j if $t_s(T_i) < t_s(T_j)$. Each transaction is assigned a priority on the basis of its timestamp: the older a transaction is, the higher priority it has.

When a transaction T_i requests a lock on a data item that conflicts with the lock currently held by another active transaction T_j, two algorithms are possible:

1. *Wait-die* (or non-preemptive technique): Therefore, an older transaction *waits* only for a younger one, otherwise the younger *dies*.
2. *Wound-wait* (or preemptive technique): Older transaction *wounds* a younger one to take its lock, otherwise the younger waits for the older one.

In both cases when an aborted transaction T_i is restarted, it has the same priority it had originally.

Both the *Wait-Die* and *Wound-Wait* methods do not create deadlocks.

When the methods need to choose between two transactions which one to abort, it is always preferred the **younger one**, because it has probably *done less work*, and therefore it is less costly to abort. Moreover, the aborted transaction is *restarted* again with the *same value* of the time stamp, and therefore it sooner or later will become the oldest transaction and will not be interrupted.

However, the behavior of the two methods is very different:

- *Wound-wait* a transaction T can wait for data locked by an older transaction or restarts a younger one T_y. Most likely is the *first* case followed by the waiting rather than restarting
- *Wait-die* a transaction T can wait for data locked by a younger transaction or it restarts due to an older one T_0. Most likely is the *second* case and then the method favors the restarting rather than waiting

25.5 Serializability without Locking

The transactions can commit at any time without requesting permission. For this reason, the methods are called **pessimistic** because are based on the idea that a bad thing is likely to happen.

Other methods, called **optimistic**, have been studied for concurrency control which, instead, do not lock data because are based on the idea that bad things are not likely to happen.

Snapshot Isolation

- A transaction can perform any database operation without requesting permission, and taking advantage of *multiversions* of each data item
- The transactions must request permission to commit
- The method is called **optimistic**

25.6 Multiple-Granularity Locking

The concurrency control techniques seen so far, based on the idea of a single record lock, is not sufficiently general to treat transactions that operate on collections of records.

Other techniques based their idea on the data to lock can have different granularities and among them is defined an inclusion relationship.

- The inclusion relation between data can be thought of as a tree of objects where each node contains all its children
- If a transaction gets an *explicit* S or X lock on a node, then it has an implicit lock in the same lock mode on all the descendants of that node.

> Multiple-granularity locking requires that before a node is explicitly locked, a transaction must first have a proper intention lock on all the ancestors of that node in the granularity hierarchy.

The intention lock types are the following:

- *IS (intentional shared lock)* allows requestor to explicitly lock descendant nodes in S or IS mode
- *IX (intentional exclusive lock)* allows requestor to explicitly lock descendants in *S, IS, X, IX* or *SIX* mode
- *SIX (shared intentional exclusive lock)* implicitly locks all descendants of node in *S* mode and allows requestor to explicitly lock descendant nodes in *X, SIX*, or *IX* mode

To deal with multiple granularity locks, it is necessary **extend** the *Strict 2PL protocol* with new rules, obtaining the protocol called *Multi-granularity Strict 2PL*:

1. A node can be **locked** by a transaction T_i in S or IS mode only if the parent is locked by T_i in IS or IX mode
2. A node can be **locked** by a transaction T_i in X, IX or SIX mode only if the parent is locked by T_i in SIX or IX mode

25.7 Locking for Dynamic Databases

Transactions can also insert or delete records into tables with indexes. These possibilities raise new problems to be solved to ensure serializability during the concurrent execution of a set of transactions, while continuing to adopt the Strict 2PL protocol.

Insertion and Deletion

The inserted or deleted records are called *phantoms* because they are records that appear or disappear from sets, that is are invisible only during a part of a transaction execution.

Concurrency Control in B+–trees

The concurrent use of a *B+–tree* index by several transactions may be treated in a simple way with *Strict 2PL protocol*, by considering each node as a granule to lock appropriately.

A better solution is obtained by exploiting the fact that the indexes are used in a particular way during the operation.

Chapter 26

Distribute Concurrency Control

Distributed concurrency control ensures the correct execution of operations that affect data that are stored in a distributed fashion on different database servers.

Specifications of concurrency control protocols use the term **agent** for each stakeholder participating to the protocol.

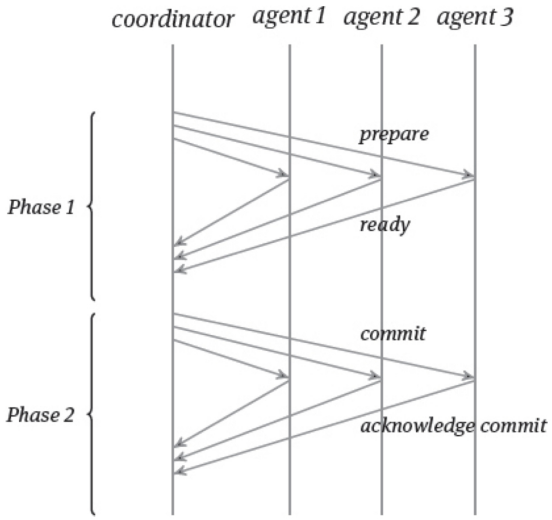
- Each agent can furthermore have different **roles**
- The most important is the **coordinator** which is the database server that communicates with all other agents
- Concurrency protocols hence offer a solution to the **consensus problem**: the consensus problem requires a set of agents to agree on a single value.

- The **two-phase commit**requires *all participating* agents to agree to a proposed value in order to accept the value as the currently globally valid state among all agents
- In contrast, **quorum consensus** protocols only require a certain *majority* of agents to agree on a proposed value
- The **Paxos algorithm** as a prominent and widely used quorum consensus protocol
- Lastly, **multi-version concurrency control** as a timestamp based concurrency mechanism that offers non-blocking reads

26.1 Two-Phase Commit

- The two-phase commit (2PC) addresses the execution of a *distributed transaction* where all agents have to acknowledge a successful finalization of the transaction
- 2PC is initiated by the coordinator of the transaction who wants to reach a consensus
- In the simplest case, all agents try to agree on accepting a single value of an update request that has been received by the coordinator

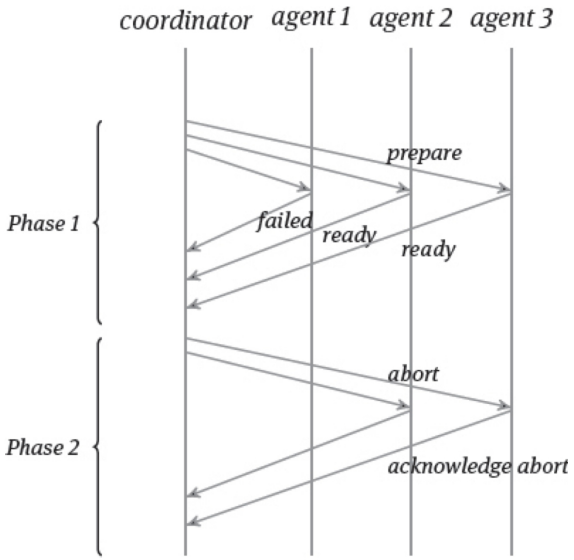
It has two phases: **voting phase** and a **decision phase**:



- In each phase, the coordinator sends one message to all agents and receives a reply from each agent
- In case no timeouts and no restarts occur, the agents can either jointly agree to commit the value or the coordinator decides to abort the transaction
- **Voting phase:** the coordinator sends all agents a **prepare** message asking f they can commit the transaction. Each agent can:
  - Replay *ready*
  - Replay *failed*
  - Do not replay, causing the time out protocol handling this case
- **Decision phase**, the coordinator notifies the agents of a common decision resulting from the votes:
  - The transaction can only be **globally committed** if all agents voted *ready* in which case the coordinator sends a commit message to all agents
  - The **abort** case applies if at least one agent voted *failed*. In order to abort the transaction globally, the coordinator has to send an *abort message* to all agents that have voted *ready*

A major problem is that a failure of a single agent will lead to a global abort of the transaction. Moreover, the protocol highly depends on the central role of a single coordinator. In particular, the state between the two phases is called the **in-doubt state**:

- In case the coordinator irrecoverably fails before sending his decision, the agents cannot proceed to either commit or abort the transaction
- It can only be solved by a complex recovery procedure that contacts all other agents and asks for their votes again



A so-called three-phase commit protocol adds another phase (the pre-commit phase) to avoid this blocking behavior

26.2 Pasox Algorithm

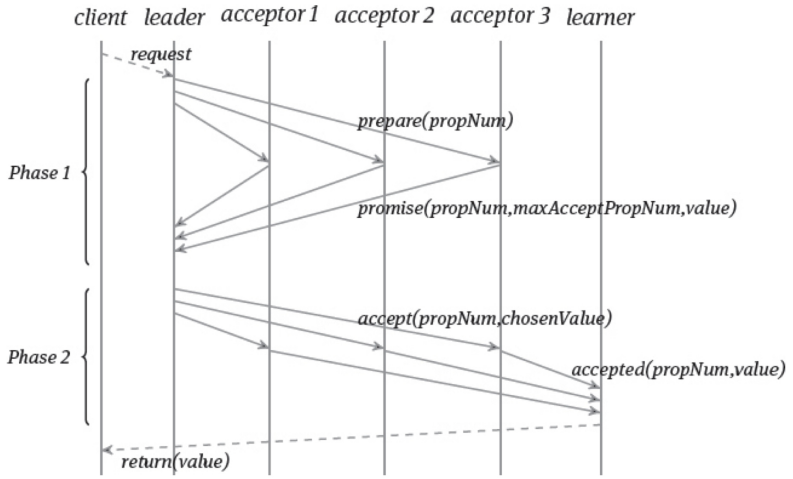
The basic Paxos algorithm is meant to cope with non-Byzantine failures and it can be applied to keep a distributed DBMS in a consistent state.

A client can for example issue a read request for some database record; the database servers then have to come to a consensus on what the current state of the record is.

The following types of agents take part in a Paxos protocol:

- **Proposer:**
  - A proposer is an agent that waits for a client request and then initiates the consensus process
  - A proposer assigns a number to its request
  - Depending on the answers received from the acceptors, the proposer chooses a response value
- **Leader:** for handling a specific client request, one of the proposers is elected to be the leader
- **Acceptor:**
  - An acceptor can accept a proposal based on the proposed value and on the proposal number
  - Correct acceptor only accepts proposals which are numbered higher than any proposal it has accepted before
  - So acceptor has to always remember the highest proposal it accepted so far
- **Learner:**

- Is any other agent that is interested in the value on which the acceptors agreed
- Learners will be informed of a response value by each of the acceptors
- Usually, the leader is also a learner so that he receives the notification that his chosen value was finally agreed to by a majority of acceptors



With the basic Paxos protocol the following safety properties are guaranteed to hold for each individual run of the protocol:

- **Nontriviality:** Any value that a learner learns must have been proposed by a proposer
- **Stability:** A learner learns one single value or none at all
- **Consistency:** All learners learn the same value
- **Liveness:** If some value has been proposed, a learner will learn a value

The basic Paxos protocol can support non-Byzantine failures of the participating agents as follows:

- **Failures of proposers:** The leader can fail as long as there is at least one backup proposer who can be the new leader and eventually gets his chosen value accepted
- **Failures of learners:** If all learners fail, the consensus value will not be sent to the client although a majority of acceptors accepted a chosen value. Hence, there must be at least one learner working correctly
- **Failures of acceptors:** The leader has to receive promise messages for the proposal number from a majority of acceptors and later on at least one learner has to receive accepted messages for the proposal number from a majority of acceptors

If however a majority of acceptors fails, a new run of Phase 1 has to be started with a higher proposal number and with a quorum containing other acceptors than the failed ones.

26.3 Vector Clock

The strong clock property is satisfied for **vector clocks**: a vector clock is a vector of counters with one counter for each client process. With vector clocks it is crucial to have a separate counter for each client processes as otherwise servers could not accept concurrent write requests from multiple users.

26.4 Version Vectors

> While vector clocks are a mechanism for stepping forward the time in a messagepassing system, version vectors are a mechanism to consolidate and synchronize several replicas of a data record.

- In order to determine which view of the state of the database contents each user has, for every read request for a data record the answer contains the current version vector for that data record
- When subsequently the user writes that same data record, the most recently read version vector is sent with the write request as the so called **context**
- With this context, the database system can decide how to order or merge the writes
- Conflicting versions can for example occur with multi-master replication, this is the case of conflicting writes
- A **synchronization** process reconciles conflicting replicas
- A simple form of synchronization of two conflicting replicas is to take the *union* of them
- The process of maintaining version vectors is slightly different from the one for maintaining vector clocks. The aim is to reconcile divergent replicas into one common version

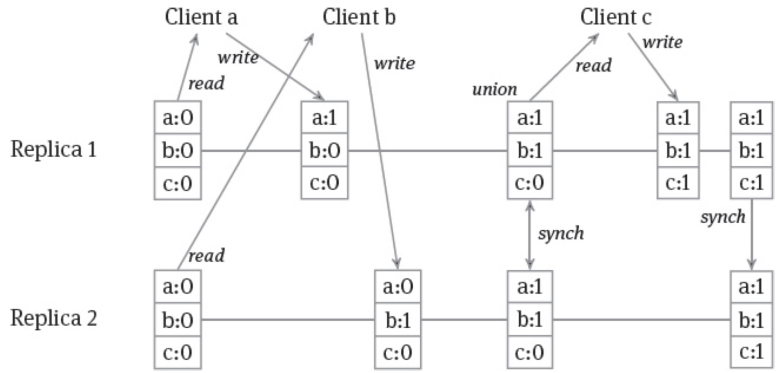
We now describe version vector maintenance with **union semantics** for the synchronization process. In this setting, each data record consists of a set of values and the synchronization of the data record computes the set union; that is, the result of a merge is again a set of values. And it is composed by the following steps:

1. **Initialization:**
   - For **n** client processes, a version vector is a vector of **n** elements
   - Each replica of a data record maintains one version vector
   - Initially, for each process all elements are 0
2. **Update:**
   - When a client process j sends a write request to overwrite a set of values at replica i, it sends the new set of values and the context

- Based on the context, it decide to overwrite its value set or take the union
- Then computes the maximum over the context and its own version vector

3. **Synchronization:**

   - Whenever two replicas i and j have different version vectors
   - The synchronization process reconciles the two versions by either overwriting one value set or by taking the union of the two value sets



More advanced forms of merging usually involve a semantic decision that requires interaction of the user or a more intelligent application logic. If such a user interaction is needed, **sibling versions** for a data record have to be maintained until the user writes a merged version:

- The database systems stores all concurrent versions together with their attached version vectors
- As soon as a user writes a merged version that is meant to replace the siblings, the version vector of the merged version is set to be larger than all the version vectors of the siblings
- And then the siblings can be deleted

More formally, version vector maintenance with sibling semantics works as follows:

1. **Initialization:**

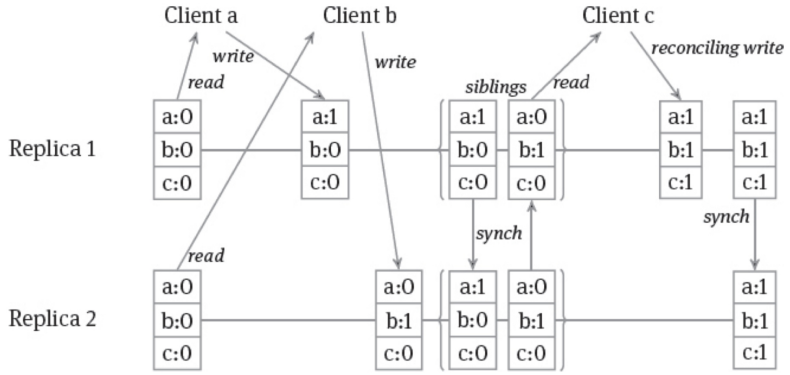
   - For **n** client processes, a version vector is a vector of **n** elements
   - Each replica of a data record maintains a set D of pairs of values and version vectors

2. **Update:**

   - When a client process j sends a write request to overwrite some or all values in the data set D_i at replica i, it sends the new value val_j and the context C_j
   - The replica checks which siblings in D_i are covered by C_j, then computes the maximum over the context

3. **Synchronization:**

   - Whenever two replicas i and j have different data sets D_i and D_j for one data record, the synchronization process reconciles the two versions by only keeping the values with the highest version vectors



26.5 Optimizations of Vector Clocks

Distributed systems with vector clocks thus need some kind of **vector clock bounding** in order to support a large number of client processes over a long period of time.

26.5.1 Client IDs versus replica IDs

- For version vectors, one way to reduce the size of the vectors is to use **replica IDs** instead of client IDs.
- This is based on the observation that synchronization takes place only between replicas and hence version vectors only need one element per replica for each data record
- However, with a simple counter for each of the replica IDs we run into the following problem of **lost updates**
- Two clients might concurrently write to the same replica based on an identical context
- With a version vector based on client IDs, the database could simply handle the stale write as concurrent
- However, with version vectors simply based on replica IDs, this concurrency cannot be expressed

Chapter 27

Consistency

- **Consistency:**
  - When a server modifies a value in one replica, the value must be immediately modified in all replicas
  - Costly for great number of replicas
  - Might block a read access until all replicas are updated
- **Availability:**
  - High performance of query answering
  - Very low response time
- **Partition Tolerance:**
  - Failures in distributed systems do not lead to failure of the entire system

CAP conjecture

Consistency, **A**vailability, **P**artition Tolerance: *Pick any two!*

27.1 Strong Consistency

After update, any subsequent access returns the updated value. However we can observe **stale read** and **lost update**.

27.2 Weak Consistency

- Ensuring strong consistency is usually costly
- Strong consistency requires a high amount of synchronization between replicas
- **Weak consistency** can improve performance of the overall system
- However, weak consistency can cause conflicts and inconsistencies of the data and may even lead to data loss

Inconsistency window

The duration from the acceptance of a write request by the first replica until the last successful write execution by all replicas is called the **inconsistency window**. Ideally, the inconsistency window is only very short in order to reduce the amount of stale reads.

Eventual consistency

- Replicas may hold divergent versions of a data item due to concurrent updates and propagation delays
- Replicas agree on a version after some time of inconsistency and as long as no new updates are issued by users
- Eventual consistency requires the two properties:
  1. Total propagation of writes to all replicas
  2. Convergence of all replicas towards a unique common value

27.3 Write and Read Quorums

A flexible mechanism to avoid stale reads and lost updates among a group of servers in a replicated database system is to use quorums when reading and writing data:

- *Read quorum* is a subset of replicas that have to be contacted when **reading data**
  - For a **successful read**, all replicas in a read quorum have to return the *same answer value*
- *Write quorum* is a subset of replicas that have to be contacted when **writing data**
  - For a **successful write**, all replicas in the write quorum have to *acknowledge the write request*

A usual requirement for a quorum-based system is that any read and write **quorums overlap: the sum of read quorum size and write quorum size are larger than the replication factor**. In this way, it can be ensured that at least one replica has acknowledged all previous writes and hence is able to return the most recent value.

Quorum rules for N **replicas**, **read** quorum size R, **write** quorum size W:

- $R + W > N$ for *consistent read*
- $2W > N$ for *consistent write*

We typically have two typical variants of quorum-based systems:

- **Read-one write-all (ROWA):** writes are acknowledged by all replicas, but for reads it suffices to contact one replica

- **Majority Quorum:** or both reads and writes requires that (for N replicas) at least $\frac{N}{2} + 1$ replicas acknowledge the writes and at least $\frac{N}{2} + 1$ replicas are asked when reading a value

When all replicas in a read quorum return an identical value, the requesting client can be sure that this is the most recent value and hence the read value is strongly consistent.

We mention also that **Partial quorums** only ensure weak consistency (for example, during failures) or when strong consistency is not required.