

Key-Value (K-V) Systems



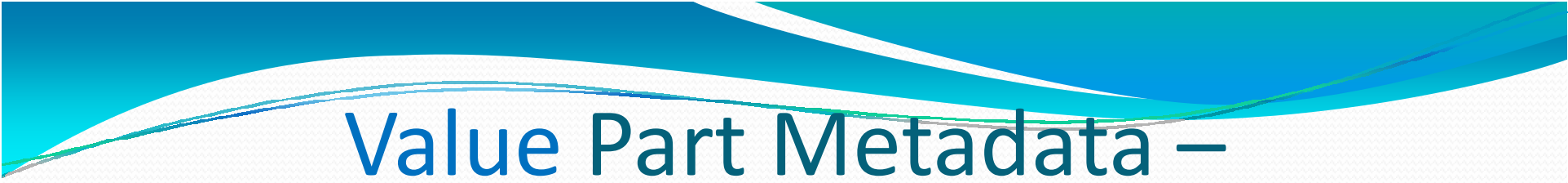
Outline

- Data design and storing
- Data querying - retrieval
- Updating the data



Data Design in K-V Systems

- A K-V store record has two parts:
 - **Key** part
 - **Value** part - string
- use XML or JSON to store **Value** part metadata (the structure of **Value**)



Value Part Metadata – *Relational Database Style*

The metadata describes for each **table** of a database:

- key part (primaryKey tag),
- the structure of the value part (Attribute tags)
- index files (IndexFile and IndexAttributes tags)
- the relationships (foreignKey tags)

Value Part Metadata – *Relational Database Style* Example in XML- *Tables*

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Databases>
- <DataBase dbName="Premium">
- <Tables>
+ <Table tableName="geo" fileName="geo.kv" rowLength="114">
+ <Table tableName="poi" fileName="poi.kv" rowLength="300">
</Tables>
- <!--
    there can be users with permissions or replication or
    views or other database objects
-->
</DataBase>
<!-- can be more databases in one xml structure file -->
</Databases>
```

Value Part Metadata – *Relational Database Style*

Example in XML - *Table's Structure*

```
- <Table tableName="geo" fileName="geo.kv" rowLength="114">
- <Structure>
  <!-- attributeName, type can be elements instead of attributes in xml -->
  <Attribute attributeName="poi_id" type="char" length="64" isnull="0" />
  <Attribute attributeName="latitude" type="double" isnull="0" />
  <Attribute attributeName="latitude" type="double" isnull="0" />
</Structure>
+ <primaryKey>
+ <IndexFiles>
</Table>
- <Table tableName="poi" fileName="poi.kv" rowLength="300">
- <Structure>
  <!-- attributeName, type can be elements instead of attributes in xml -->
  <Attribute attributeName="poi_id" type="char" length="64" isnull="0" />
  <Attribute attributeName="language" type="int" isnull="0" />
  <Attribute attributeName="poiName" type="char" length="255" isnull="0" />
</Structure>
+ <primaryKey>
+ <foreignKeys>
+ <uniqueKeys>
+ <IndexFiles>
</Table>
```



Value Part Metadata – *Relational Database Style*

Example in XML - *Table's Structure*

- `<!-- attributeName type can be element instead of attribute in xml -->`

Value Part Metadata – *Relational Database Style*

Example in XML - *Primary Key*

```
- <Table tableName="geo" fileName="geo.kv" rowLength="114">
+ <Structure>
- <primaryKey>
  <pkAttribute>poi_id</pkAttribute>
  <!-- can be more attributes in primary key -->
</primaryKey>
+ <IndexFiles>
</Table>
- <Table tableName="poi" fileName="poi.kv" rowLength="300">
+ <Structure>
- <primaryKey>
  <pkAttribute>poi_id</pkAttribute>
  <pkAttribute>language</pkAttribute>
</primaryKey>
+ <foreignKeys>
+ <uniqueKeys>
+ <IndexFiles>
</Table>
```


Value Part metadata – *relational database style*

Example in XML - *foreign key*

```
- <Table tableName="poi" fileName="poi.kv" rowLength="300">
+ <Structure>
+ <primaryKey>
- <foreignKeys>
  - <foreignKey>
    <fkAttribute>poi_id</fkAttribute>
    <!-- can be more attributes in foreign key -->
  - <references>
    <refTable>geo</refTable>
    <refAttribute>poi_id</refAttribute>
    <!-- can be more attributes in referenced key -->
  </references>
  </foreignKey>
</foreignKeys>
+ <uniqueKeys>
+ <IndexFiles>
</Table>
```

Value Part Metadata – *Relational Database Style*

Example in XML – *Index Files*

```
- <Table tableName="poi" fileName="poi.kv" rowLength="300">
+ <Structure>
+ <primaryKey>
+ <foreignKeys>
+ <uniqueKeys>
- <IndexFiles>
  <!-- for primary keys there can be index files by default -->
  - <IndexFile indexName="poi_poid.ind" keyLength="64" isUnique="0" indexType="BTree">
    - <IndexAttributes>
      <IAttribute>poi_id</IAttribute>
      <!-- can be more attributes in index key -->
      <!-- attributes can be referenced by position too -->
    </IndexAttributes>
  </IndexFile>
  - <IndexFile indexName="poiName.ind" keyLength="255" isUnique="1" indexType="BTree">
    - <IndexAttributes>
      <IAttribute>poiName</IAttribute>
    </IndexAttributes>
  </IndexFile>
</IndexFiles>
</Table>
```



Data Design-

Relational Database Style

For each table T in the relational DB the following key-value files are produced:

- T data are stored in a master K-V file T.K-V
 - **Key** = T.PrimaryKey
 - **Value** = concatenation of T non-primary key attributes (columns)
- for every unique index file T.I of T a new T.I.K-V file is created
 - **Key** = concatenation of IAttribute values
 - **Value** = T.key (master)
- for every non-unique index file T.I of T a new T.I.K-V file is created

Master K-V, Unique Index K-V File

Example: poi Table

1. Master K-V file `poi.kv`

Key = `poi_id + language`

Value = concatenation of poi non-primary key attributes

2. Unique Index `poiName` in K-V file `poiName.ind`

Key = `poiName`, (e.g. “La Scala di Milano”)

Value = `(poi_id + language)`



Non-unique Index File

- **Solution 1: *Inverted index.***
- **Solution 2: *Make search key unique***



Non-unique Index File - *Inverted Index*

- A single K-V record for each search key is built
 - **Key** = search key value
 - **Value** = list of tuple pointers of the records (primary keys from master), where the corresponding search key is in the value part.

Example: latitude is a search key for geo

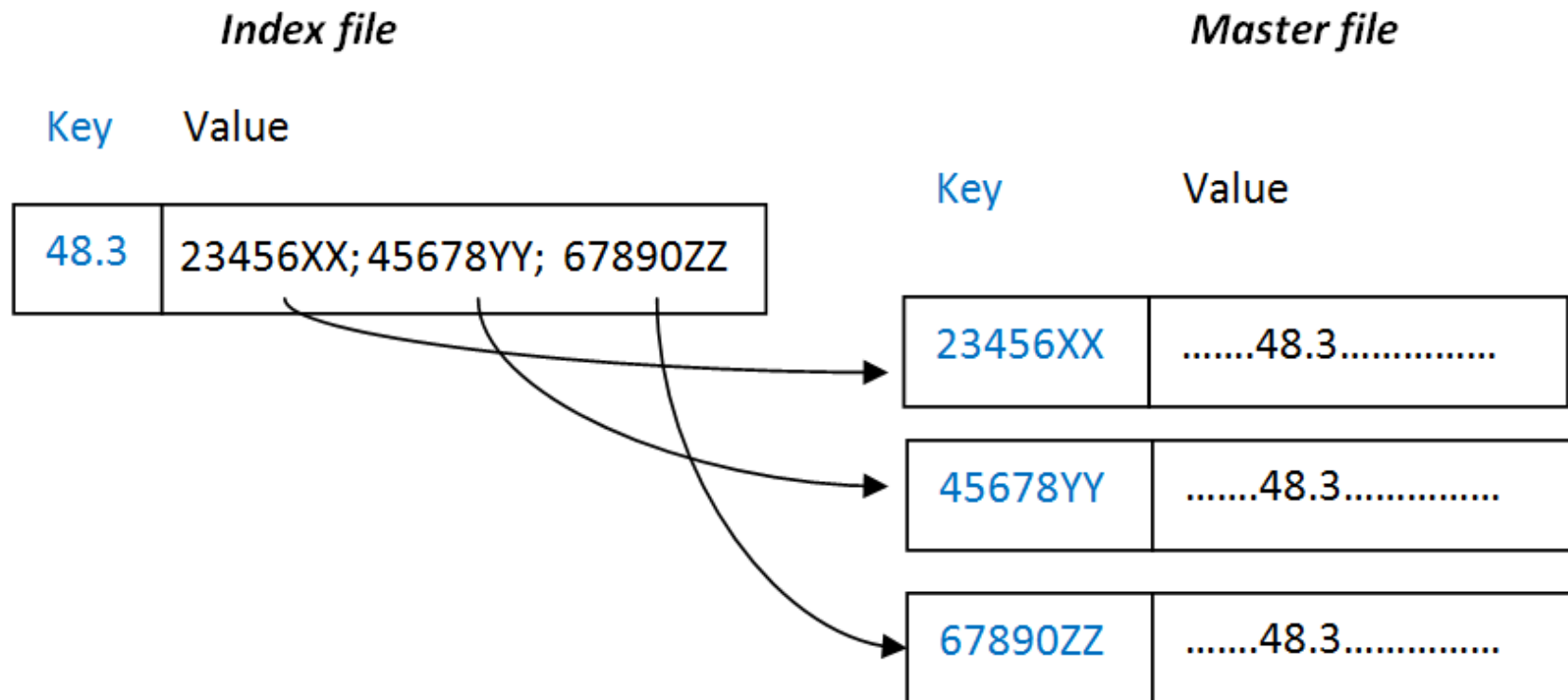
Let be 3 poi_id: 23456XX, 45678YY, 67890ZZ for which the latitude is 48.3.

Key = 48.3

Value = 23456XX;45678YY;67890ZZ

(48.3, (23456XX, 45678YY, 67890ZZ)) → Inverted index

Inverted Index- Example





Non-unique Index File - *Inverted index* *Implementation Details*

- Extra code to handle long lists
- Deletion of a record can be expensive
- Low space overhead, no extra cost for queries



Non-unique Index File – ***Make Search Key Unique***

- For each search key many K-V records are built, one for each record in the master with the same search key value.
- This is done by making search key unique, adding a record-identifier.

Key = (search key + record identifier)

Value = NULL.



Non-unique Index File – *Make Search Key Unique* - Example

For the example above, 3 different K-V records are built:

(48.3#23456XX, NULL);

(48.3#45678YY, NULL);

(48.3#67890ZZ, NULL);

- The “#” is used to separate the key part from unique identifier.
- The use of a separator can be avoided if constant (maximum) key-length is implemented. Shorter than maximum length keys are padded with spaces/blanks.



Non-unique Index File – *Make Search Key Unique* Implementation Details

- it is widely used for indexing in relational databases;
- in order to implement it for a K-V store, it needs:
 - extra storage overhead for keys
 - extra custom code for insertion/deletion operations

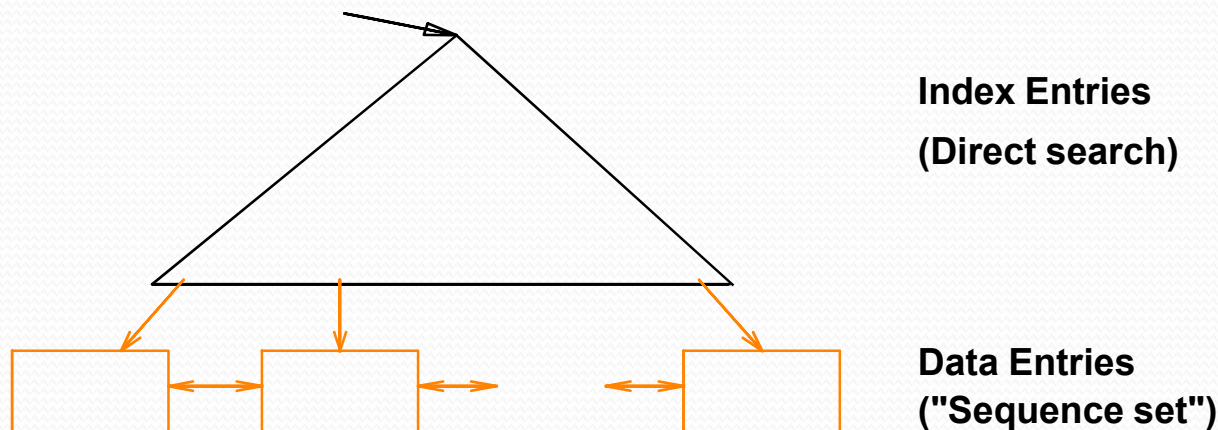


Performance Problems of Solutions for Index Files

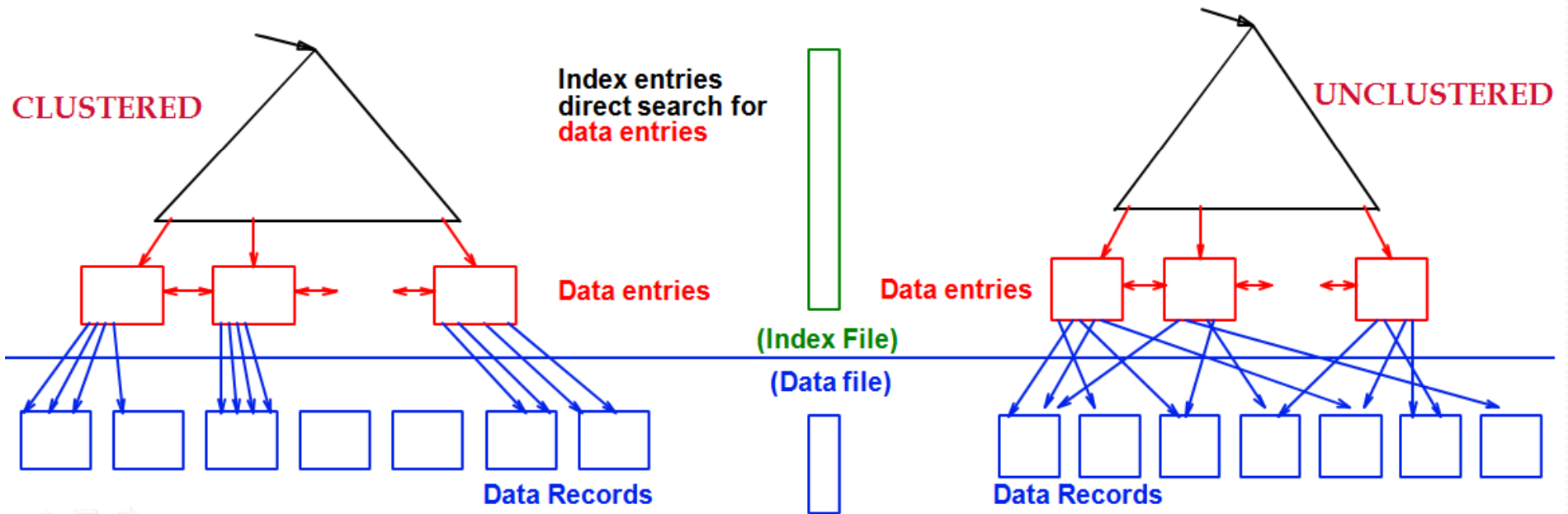
- in order to find a record using index files a supplementary read operation is needed compared to a relational index:
 - Read from K-V index file – return one or more primary keys
 - Read from the K-V file by primary key – depends on the index behind: **hash (only equality search)** or **B+Tree (range search possible)**
- **Remark.** In relational database systems, an index file record contains the search key and the physical address(es) of the data record(s) on the disk, so the search needs only one block access from the data file.

B+ Tree: The Most Widely Used Index

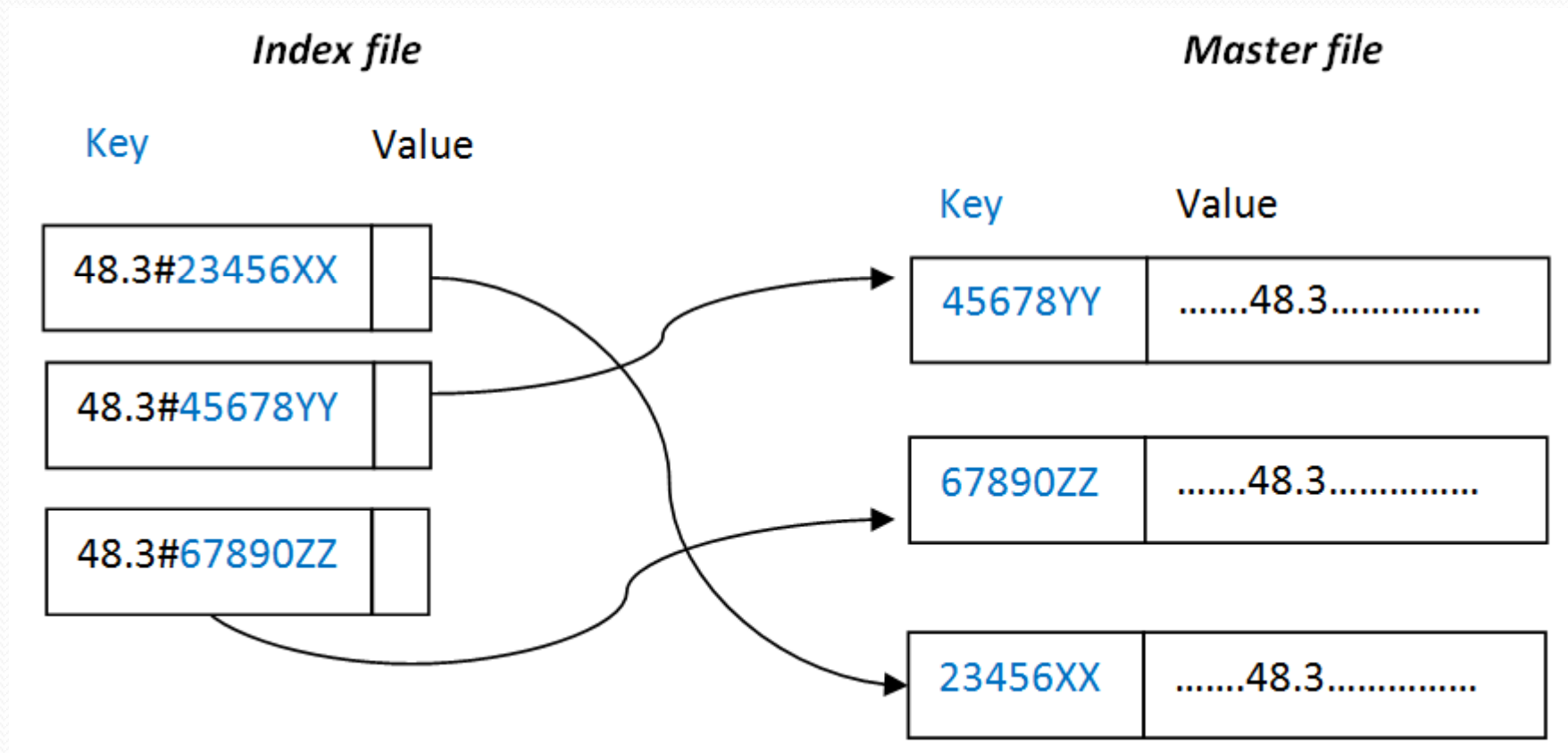
- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*.
 - F = fanout, N = # leaf pages
- minimum 50% occupancy (except for root)
- each node contains m entries where $d \leq m \leq 2d$ entries
- d is called the *order* of the tree.
- supports equality and range-searches efficiently.
- all searches go from root to leaves, but structure is *dynamic*.



Clustered vs. Unclustered Index



Solutions for Index Files





Querying the Key-Value System

Relational Database Style

- a **general** solution: implement the SELECT statement over a K-V system - see [SKS06], [RG03]
- how to transform one-table queries and multiple-table queries, respectively in operations on K-V system
- a custom solution for spatial queries is proposed



Querying One Table

- Search by unique key – reduces to using K-V `get` method on master and index files.
- Search by non-unique key –
 - **Solution 1: *Use of inverted indexes***. The programmer has to handle the list of primary keys from K-V index files and retrieve every record from K-V master with the searched key, using `get`.

Querying One Table

Search by Non-unique Key

Solution 2. Make search key unique by adding a record-identifier.

- range queries in K-V systems are not implemented yet ([Voldemort10], [RRHSo4])
- B+ Tree index is suitable for range queries, if the K-V system allows direct access to the tree.



Querying One Table

Search by Non-unique Key

Solution 2. Make search key unique by adding a record-identifier.

Possible solutions:

- design an additional API in Voldemort for range queries, use the B+ tree from DB layer.
- the proposed layered architecture is not strict, allowing the Java client to access directly the B+ tree from DB layer
- the programmer has to implement B+ Tree ([SKS06]) or prefix hash tree ([RRHS04])

Querying One Table

Search by Non-unique Key

Solution 2. Make search key unique by adding a record-identifier.

Example: The search of all poi_id-s for latitude 48.3 is transformed to the following range query:

`minKey=48.3# + possible min value`

`maxKey=48.3# + possible max value`

`minKey <= key <= maxKey`



Queries involving More Tables

- each specific query is mapped in a custom way.
- some **general guidelines** can be given:
- in order to fetch as few data as possible in the memory, the first operation is to apply the filter for each table.



Queries involving More Tables -Example

- apply the filter for each table, which reduces to a one-table query
- compute the join in memory if possible; if not, apply hash join or other join types.
- compute GROUP BY operation of the query
- compute sort operation of the query
- optimization issues are not covered here.
([RG03], [MUW08], [SKS06]).



References

- [BK92] Elisa Bertino, Won Kim: *Indexing Techniques for Queries on Nested Objects*, IEEE Transactions on Knowledge and Data Engineering, Volume 1 Issue 2, June 1989
- [LOH92] C. C. Low; B. C. Ooi, H. Lu: *H-trees: a dynamic associative search index for OODB*, Proceeding SIGMOD '92 Proceedings of the 1992 ACM SIGMOD international conference on Management of data.
- [MUW08] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: *Database Systems: The Complete Book*, Prentice Hall, 2008.
- [RG03] R. Ramakrishnan, J. Gehrke: *Database Management Systems*, Third Edition, WCB McGraw-Hill, 2003.
- [RRHS04] Sriram Ramabhadran , Sylvia Ratnasamy , Joseph M. Hellerstein , Scott Shenker: *Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables* (2004), Technical Report <http://berkeley.intel-research.net/sylvia/pht.pdf>



References

- [SY06] Toshiyuki SHIMIZU, Masatoshi YOSHIKAWA: *Full-Text and Structural Indexing of XML Documents on B+-Tree*, IEICE TRANSACTIONS on Information and Systems Vol.E89-D No.1, 2006, pp.237-247
- [SKS06] A. Silberschatz, H. Korth, S. Sudarshan: *Database System Concepts*, McGraw-Hill, New York, 2006.
- [SC10] Michael Stonebraker, Rick Cattell: *Ten Rules for Scalable Performance in “Simple Operation” Datastores*, 2010, <http://www.cattell.net/datastores/CACM-Paper.pdf>
- [Voldemort10] Range queries in K-V systems, http://groups.google.com/group/project-voldemort/browse_thread/thread/cad4888b492d897f