Philipps University of Marburg
Department of Mathematics and Computer Science
Database Research Group

Winter semester 2024/2025
29. 11. 2024

Prof. Dr. Bernhard Seeger
Amir El-Shaikh, M.Sc.
Marius Kuhrt, M.Sc.

Exercises for the lecture
Implementation of Database Systems

Submission: 13. 12. 2024,
no later than 8:00 p.m.
via the ILIAS platform

Exercise 4

Task 4.1: Basic COLA (4+7+3+1) (15 Points)

In this exercise you will implement the basic variant of the COLA data structure. For this purpose, you can find the classes `BasicCOLA<K extends Comparable<K>,V>` and `COLABlock<K,V>` in the ILIAS platform. The elements in the COLA data structure are managed using `COLABlock<K,V>`. Additionally, you need the dependency `xxlcore-2.1.jar`, which is provided via the ILIAS platform like-wise.

The data structure is intended to hold the first levels (arrays) in main memory. All other levels are stored on external storage. A `COLABlock` has the size of a page on the external storage. This means that arrays larger than a page are mapped to multiple `COLABlock`s. For reconstruction, the offset of the first `COLABlock<K,V>` block of a level in the underlying container is stored in the `arrayOffsets` data field. In addition, the boolean list `filled` marks each array as filled or empty. The `COLALevel` interface provides useful abstraction for both memory- and disk-based levels.

a) In the `BasicCOLA` class constructor, wrap the given `Container` into a `ConverterContainer` instance. For this purpose, implement a `Converter` for `COLABlock<K,V>` that uses the two `Converter` for keys (`K`) and values (`V`). Make sure that the converter always reads and writes full blocks, even if the given block is not completely filled.

b) Implement the methods `V search(K key)` in both `CacheLevel` and `DiskLevel` to perform binary search on the elements in each level. Make sure that binary search on the `DiskLevel` only reads the necessary blocks on demand, not the full level.

c) Implement `searchElement(K key)` in `BasicCOLA` to perform a top-down search for the given key over all levels.

d) Test your implementation via the provided `main` method. The binary file `timeseries.bin` contains data in the format *<date, market-value>*. The date is stored as an *8 bytes integer*, the market value as an *8 bytes floating point*. The data is inserted into a COLA data structure and the very first and very last keys are looked up. Measure the time each lookup takes by repeating it a reasonable number of times and averaging the result.

Task 4.2: Indexing (7+10+3) **(20 Points)**

The primary index of a table directly maps **TIDs** (Tuple IDs) to their corresponding rows. A secondary index, on the other hand, maps a specific column value to a set of **TIDs**, which can then be used to retrieve the corresponding rows via the primary index.

A `Schema` defines the structure of the relation types. A `Table` is created using a `Schema` and a `PrimaryIndex`.

A `ResultSet` represents a query result on the table. Note that it does not contain actual rows, but **TIDs**. Rows are fetched from the primary index only when requested with `stream()`.

a) Complete the implementations of the classes `SecondaryHashIndex` and `SecondaryTreeIndex`. The implementation should simply wrap Javas `HashMap` and `TreeMap`, respectively.

b) Implement the following class methods in `Table`:

1. `void insert(Row row)`, inserts a row into the primary index and all existing secondary indexes.
2. `boolean remove(ResultSet set)`, removes all rows in `set` from the primary and all secondary indexes.
3. `ResultSet pointQueryAtColumn(int columnIndex, Object value)`, returns a `ResultSet` of all rows with the given `value` at the given `columnIndex`. If it exists, the query should use the existing index on the column. Otherwise, it must scan the primary index and filter the results.
4. `ResultSet rangeQueryAtColumn(int columnIndex, Object from, Object to)`, the query should use an existing index on the column if it exists *and* supports range queries; otherwise, it must scan the primary index and filter the results. The method should throw an exception if the type of the column doesn't have a comparator, i.e., if `Schema::getComparatorOfColumn` returns `null`;

c) In the `Main` class, translate the following SQL query into Java code using queries on the Table, assuming `col0` and `col1` are the names of the two columns in the schema. Print the results on the console.

```
SELECT * FROM Table
WHERE (col0 < 100 AND col1 like 'A%')
OR col0 BETWEEN 1000 AND 1010;
```

Hint: Take a look at `ResultSet`'s methods.

The `Example` class displays how `Table`s can be used (after fully implemented), along with a small benchmark on the performance of point queries using a hash/tree index.

Note: The following tasks are optional but *highly* encouraged.

Task 4.3: Linear Hashing **(0 Points)**

Given are the following parameters for linear hashing:

$$n = 5, \quad b = 2, \quad bf_s = 0.75, \quad h_j(k) = k \mod 2^j n.$$

Insert the following integers into an empty hash table:

`200, 325, 405, 389, 188, 101, 500, 57, 82, 120, 133, 48, 436, 461, 364, 223, 228, 199, 123, 42, 23`

Sketch the state of the table after every insert.

Task 4.4: Bitmap Index (2+1+2+1) **(0 Points)**

Consider the relation $R = \{10, 5, 6, 8, 7, 1, 3, 5, 6, 4\}$.

a) Build a multi-component bitmap index with basis $< 5, 4 >$ for the relation R.

b) In general, how do n and m have to be chosen so that any number can be produced?

c) Build a range-coded bitmap index for the relation R.

d) Why are two accesses needed for a point query using a range-coded bitmap index?