Instituto Superior Técnico

MSc in Computer Science and Engineering

Data Administration in Information Systems

Project

Work by:	*Number:*
Diogo Moura	86976
Xavier Gomes	90637

Group 11

2020/2021

Contents

1 SQL Server Storage and Indexing

1.a

To deal with the exponential growth of the table we decided to set the file growth to 20%, instead of a fixed value.

```
CREATE DATABASE "IST-LH-Records" ON
PRIMARY (
    NAME = istDB_File1,
    FILENAME = 'C:\Temp\istDB_File1.mdf',
    SIZE = 23MB,
    MAXSIZE = 2TB,
    FILEGROWTH = 20%),
FILEGROUP SECONDARY_1 (
    NAME = istDB_File2,
    FILENAME = "C:\Temp\istDB_File2.ndf',
    SIZE = 23MB,
    MAXSIZE = 2TB,
    FILEGROWTH = 20%)
LOG ON (
    NAME = istDB_Log,
    FILENAME = "C\Temp\istDB_Log.ldf',
    SIZE = 13MB,
    MAXSIZE = 1TB,
    FILEGROWTH = 20%);
```

1.b

Based only on the restrictions indicated in the project description, we created the tables in the following way. To be able to partition the Occurrence table based on the date, we had to include this date in the primary key of that table.

```
CREATE PARTITION FUNCTION OccurrenceRange(DATETIME)
AS RANGE LEFT FOR VALUES (N'2021-04-30T15:00:00');

CREATE PARTITION SCHEME OccurrenceScheme
AS PARTITION OccurrenceRange TO
([PRIMARY], SECONDARY_1);
```

```
CREATE TABLE HealthProfessional (
    ProfessionalId INT,
    ProfessionalName VARCHAR(255),
    ProfessionalBirthdate DATE,
    ProfessionalRole VARCHAR(255),
    ProfessionalSalary INT,
    ProfessionalEmail VARCHAR(320) UNIQUE NOT NULL,
    ProfessionalPhone CHAR(9),
    ProfessionalPerformanceEval INT,
    ProfessionalStartDate DATE,
    PRIMARY KEY(ProfessionalId));

CREATE TABLE Stock (
    StockId INT,
    StockType VARCHAR(127),
    StockService VARCHAR(127),
    StockDescription VARCHAR(255),
    StockQuantity INT,
    StockDeliveredDate DATE,
    StockNeedToOrder BIT,
    PRIMARY KEY(StockId));

CREATE TABLE Schedule (
    ScheduleId INT,
    ProfessionalId INT,
    ScheduleSlotStart DATETIME,
    ScheduleSlotEnd DATETIME,
    PRIMARY KEY(ScheduleId),
    FOREIGN KEY(ProfessionalId) REFERENCES HealthProfessional(ProfessionalId));

CREATE TABLE Patient (
    PatientId INT,
    PatientName VARCHAR(255),
    PatientBirthdate DATE,
    PRIMARY KEY(PatientId));

CREATE TABLE Occurrence (
    PatientId INT,
    OccurrenceId INT,
    OccurrenceType VARCHAR(127),
    OccurrenceDate DATETIME,
    OccurrenceHospitalization BIT,
    PRIMARY KEY(PatientId, OccurrenceId, OccurrenceDate),
    FOREIGN KEY(PatientId) REFERENCES Patient(PatientId)
) ON OccurrenceScheme(OccurrenceDate);
```

1.c

As both these cases relate to range queries, and will be very similar in terms of output, we decided to create two non-clustered covering indexes, both of them including the search value and the ids that will be shown. This will allow us to get the wanted values without necessarily accessing the physical table, providing a faster response time when performing the queries.

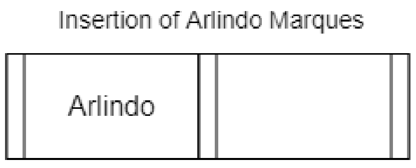
CREATE INDEX coveringRole ON HealthProfessional(ProfessionalRole, ProfessionalId);

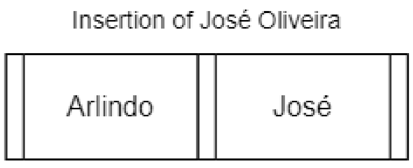
CREATE INDEX coveringAge ON HealthProfessional(ProfessionalBirthdate, ProfessionalId);

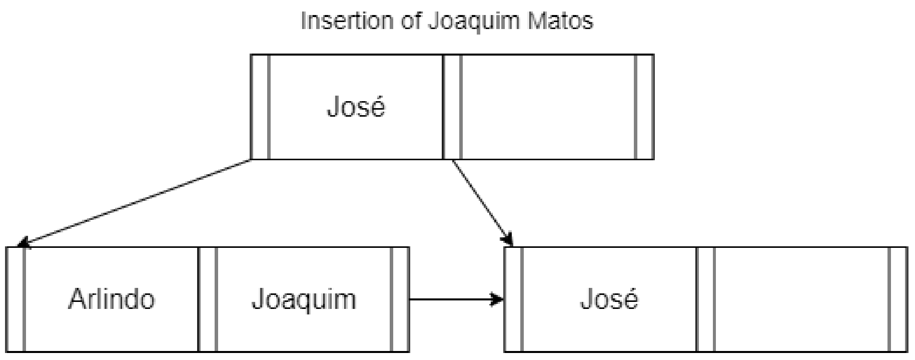
2 B+ Trees

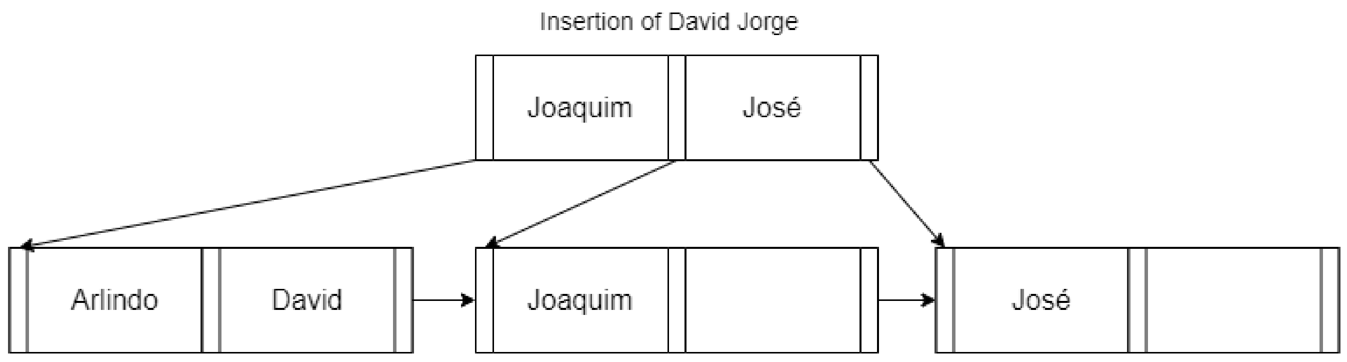
2.a

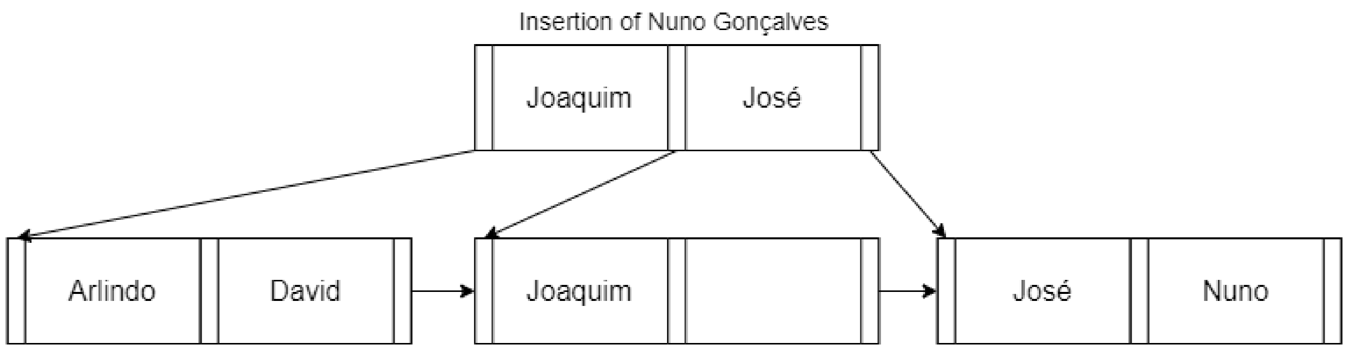
Starting with an empty B+ Tree, we got the following results after each insertion:

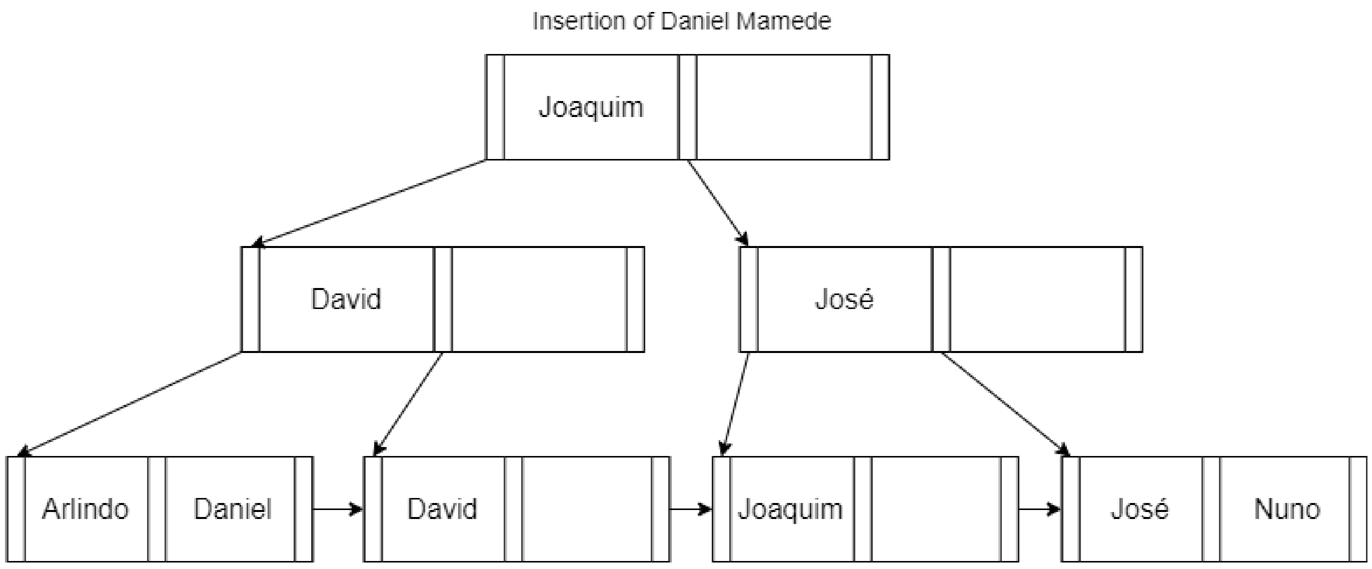






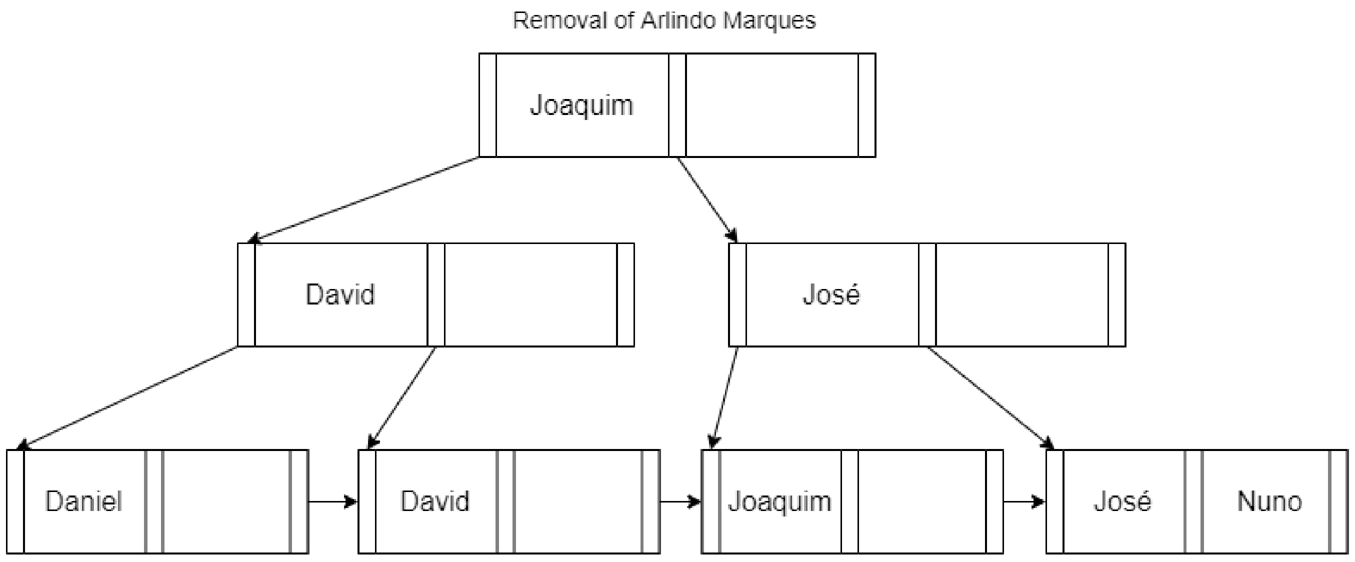

Insertion of David Jorge
Joaquim
José
Arlindo
David
Joaquim
José

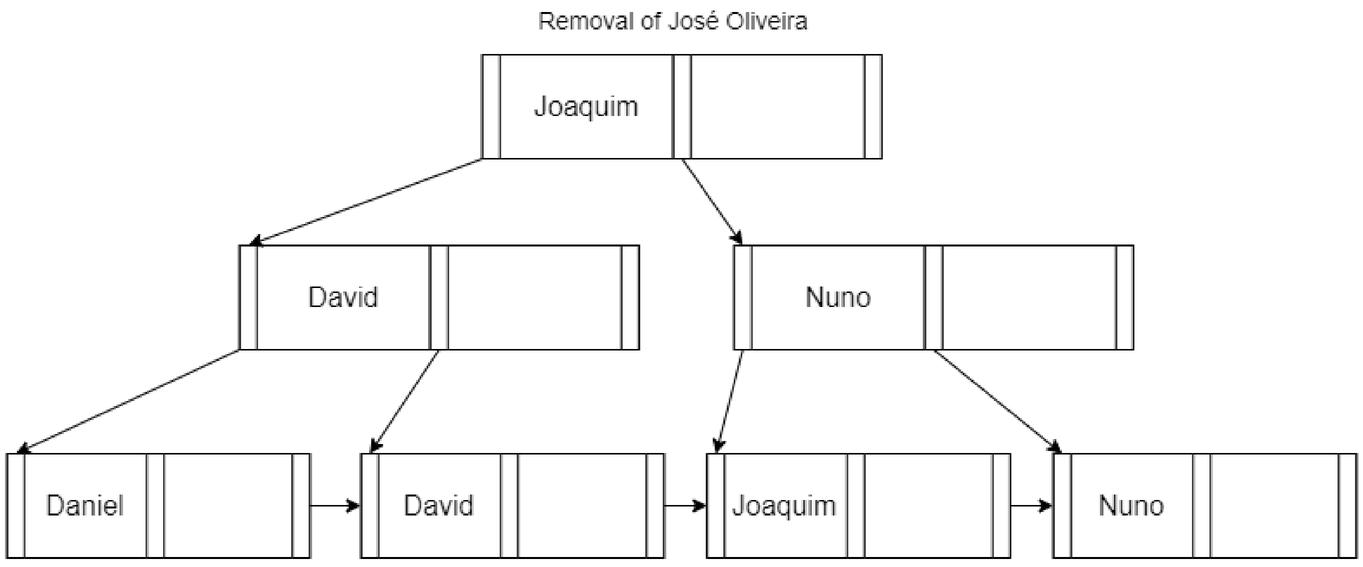

Insertion of Nuno Gonçalves
Joaquim
José
Arlindo
David
Joaquim
José
Nuno


Insertion of Daniel Mamede
Joaquim
David
José
Arlindo
Daniel
David
Joaquim
José
Nuno

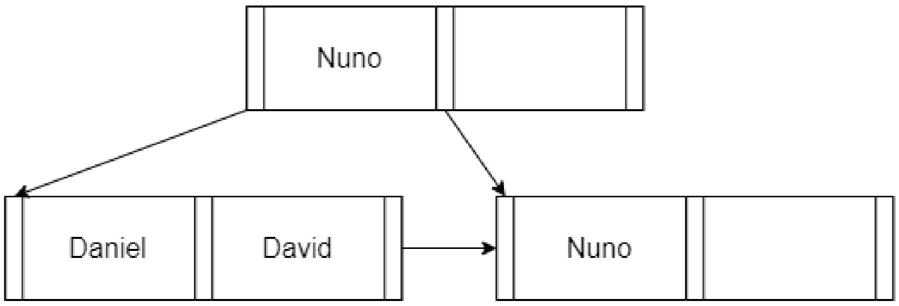
2.b

Starting from the previous question's final result, we now present the results of the B+ Tree after each deletion:

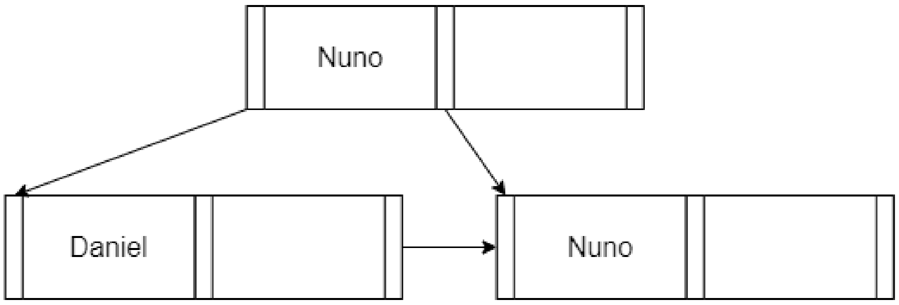




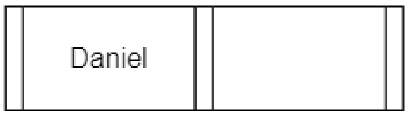
Removal of Joaquim Matos



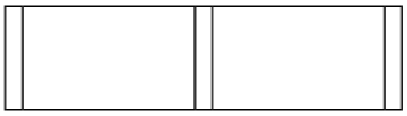
Removal of David Jorge



Removal of Nuno Gonçalves



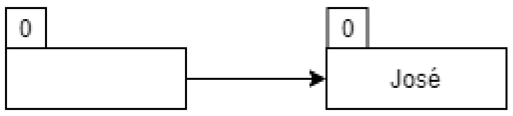
Removal of Daniel Mamede



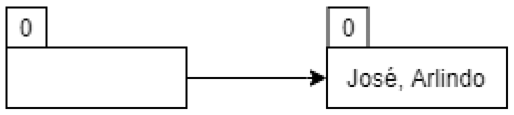
3 Extendable Hashing

Starting from an empty hash index, we got the following results after each insertion:

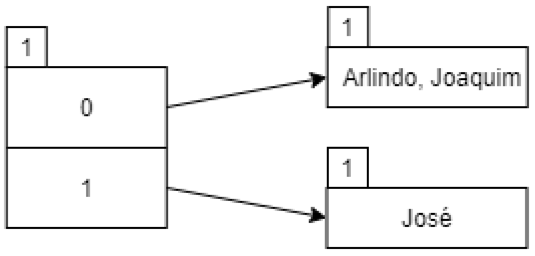
Insertion of José Oliveira



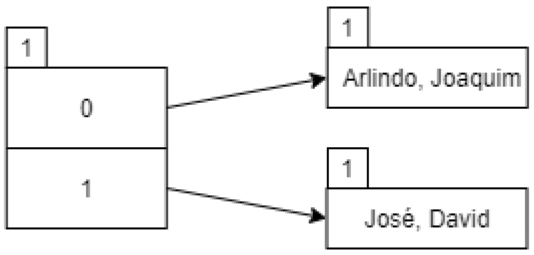
Insertion of Arlindo Marques

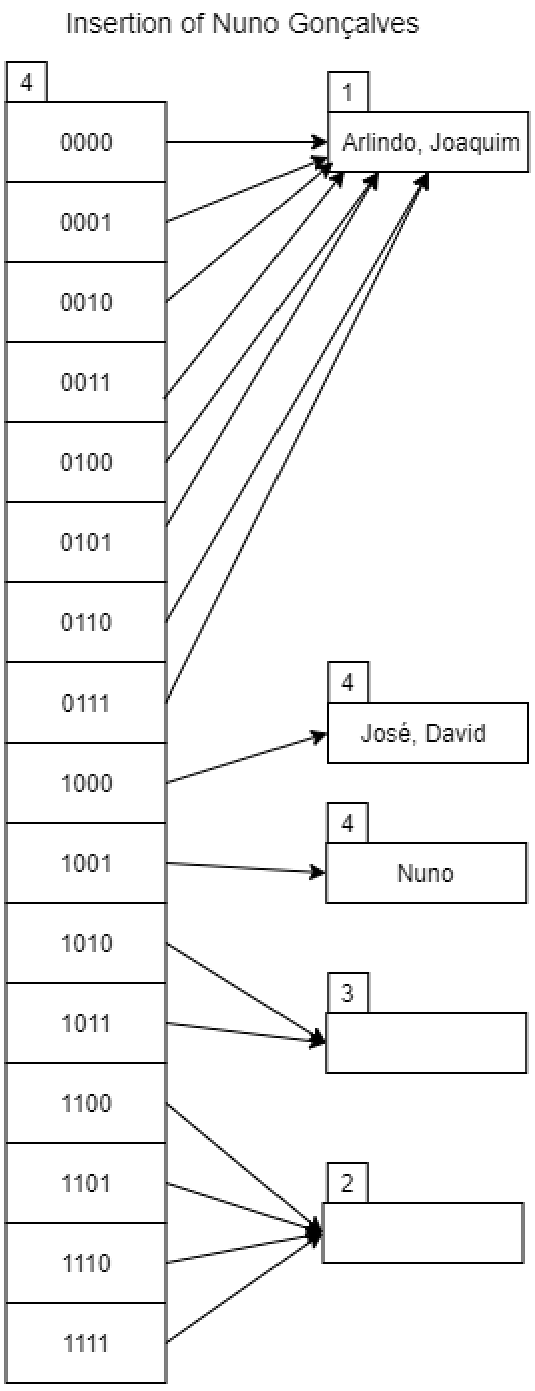


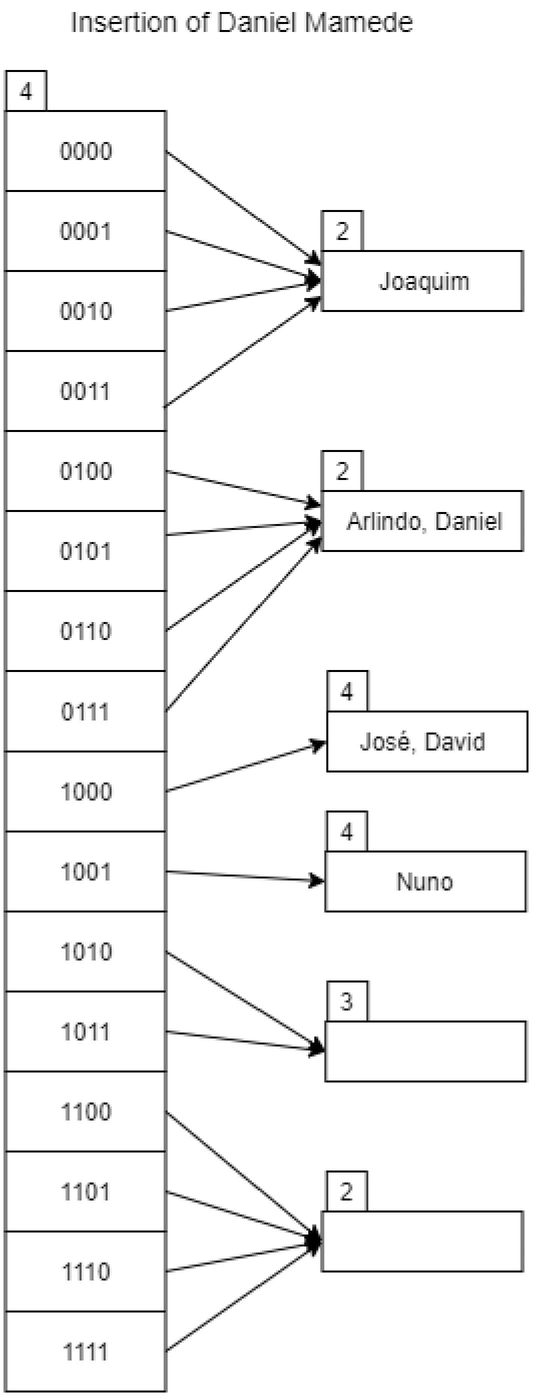
Insertion of Joaquim Matos



Insertion of David Jorge




Insertion of Nuno Gonçalves
4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
1
Arlindo, Joaquim
4
José, David
4
Nuno
3
2


Insertion of Daniel Mamede
4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
2
Joaquim
2
Arlindo, Daniel
4
José, David
4
Nuno
3
2

4 Query Processing and Optimization

4.a

With $K = 729$ search values, the tree height is no more than $ceil(log_{(ceil(n/2)}(K)) = ceil(log_3(729)) = 6$. So 6 nodes are accessed going down the tree. So 6 pages are accessed going down the tree, plus 132 records with each 27 records accounting for one page. So the final result is 6+ceil(132/27)=11 pages.

4.b

4.b.1 Indexed nested loop join, using Schedule as the inner relation and using a B+tree over professionalId on Schedule. The index tree has a height of 7.

The outer relation is HealthProfessional, which has 3921 pages and 55 tuples per page.
The inner relation is Schedule, which has 24601 pages and 33 tuples per page.
The number of pages of the outer relation is b_r, the number of tuples in the outer relation is n_r and the cost of selecting one tuple is c, which is equivalent to the height of the tree plus one.

$$Cost = b_r + n_r \times c = 3921 + (3921 \times 55) \times (7 + 1) = 1729161.$$

4.b.2 Hash-join assuming that you have 30 pages of memory and a fudge factor of 1.7. Indicate the best probe relation.

Use HealthProfessional as the build relation because it is smaller and the Schedule table as the probe relation.
Fudge factor(f) is 1.7 and Memory(M) is 30 pages, .
Number of partitions is $n = \lceil B_{H_P}/M \rceil \times f = \lceil 3921/30 \rceil \times 1.7 = 222.7$.

As $n >$ M, recursive partition is used and the cost is given by:
$Cost = 2 \times [(b_{H_P} + b_S) \times \lceil \log_{M-1}(b_{H_P}) - 1 \rceil] + b_{H_P} + b_S \ = 2(3921 + 24601)\lceil Log_{30-1}(3921) - 1 \rceil + 3921 + 24601 = 142610$

4.c

The hospital wants to check the name of the patients that have diabetes, were hospitalized in the past and were born after 2003.
The equivalent expression in relational algebra is:
$\pi_{name}(\sigma_{hospitalization=True \wedge type=diabetes}(Ocurrence) \bowtie \sigma_{p-birth-date>31-12-2003}(Patient))$.

1. $\sigma_{hospitalization=True \wedge type=diabetes}(Ocurrence)$
   Assuming that there is the same number of patients for each year and knowing that one third of patients were hospitalized, the number of tuples resulting from the first selection, from the table Ocurrence is: $30000 \times \frac{1}{3} \times \frac{2021-2003}{2021-1921} = 1800$.

2. $\sigma_{p-birth-date>31-12-2003}(Patient)$
   Assuming each type of occurrence has the same number of occurrences, the number of tables resulting from the second selection of the table is:
   $10000 \times \frac{1}{250} = 40$.

3. Join($\bowtie$)
   Since tables are being joint on the primary key of one of them there is an upper bound on the number of tuples of the resulting set of the smaller number of tuples of the relations being joined.
   The number of tuples in the query result is: min(1800,40)=40.

5 Transactions, Concurrency Control and Recovery Management

5.a

Considering H as a Health Professional record and P as a Patient one, an example of a schedule that is possible under timestamp-based protocol (with Thomas' Write Rule) but not possible under two-phase locking is:

S: T1: Read(H)
T2: Write(H)
T1: Read(P)
T2: Read(P)

From the transactions:

T1	T2

Read(H)
Read(P)
	Write(H)
	Read(P)

This schedule would lead to a case in which, when using the timestamp-based protocol, (considering TS(T1) = 1 and TS(T2) = 2) we would get:

T1 (TS=1)	T2 (TS=2)
Read-TS(H) = 1 (OK)	
	Write-TS(H) = 2 (OK)
Read-TS(P) = 1 (OK)	
	Read-TS(P) = 2 (OK)

But when using the two-phase locking we would get:

T1	T2
Lock-S(H)	
Read(H)	`Lock-X(H)!!!`

And so T2 wouldn't be able to get the lock on H and the transaction would fail.

As for the opposite case where the schedule works for the two-phase locking protocol but not for the timestamp-based one, we could have:

S: T1: Read(P)
T2: Write(H)
T1: Write(P)
T1: Read(H)

From the transactions:

T1	T2

Read(P)
Write(P)
Read(H)
	Write(H)

And in this case we would have, for the timestamp-based protocol, using the same TS values:

T1 (TS=1)	T2 (TS=2)
Read-TS(P) = 1 (OK)	
	Write-TS(H) = 2 (OK)
Write-TS(P) = 1 (OK)	
(TS(T1) < Write-TS(H) = 2) !!!	

And this would fail even with the Thomas' Write Rule being applied, as it only ignores these cases when they occur during write operations, and in this case it did during a read, and so the transaction needs to be rolled back.

As for the two-phase locking, we would have:

T1	T2
Lock-X(P)	
Read(P)	Lock-X(H)
	Write(H)
Write(P)	Unlock(H)
Lock-S(H)	
Read(H)	
Unlock(P)	
Unlock(H)	

Completing the pair of transactions just like we expected.

5.b

We have the transactions

T3	T4
Write(t1[H])	Read(t1[H])
Write(t2[H])	Read(t2[H])

Considering TS(T3) = 3 and TS(T4) = 4 we can get the desired result with the schedule:

S: T3: Write(t1[H])
T4: Read(t1[H])
T4: Read(t2[H])
T3: Write(t2[H])

This schedule will lead us to:

	T3	T4
1	Write-TS(t1[H])=3 (OK)	
2		Read-TS(t1[H])=4 (OK)
3		Read-TS(t2[H])=4 (OK)
4	**(TS(T3) < Read-TS(t2[H])=4) !!!**	

On step 4, T3 will fail it's write operation, causing it to restart. Because T4 had read t1 on step 2 after T3 wrote it on step 1 and before it was committed, T4 will also cascade rollback as we wanted.

5.c

To help with showing the contents of the requested we start by parsing the simplified representation for the log file into a more helpful table for us:

ATT - Active Transaction Table
DPT - Dirty Page Table

LSN	Action
20	Add (T1, 20) to ATT, Add (P1, 20) to DPT
30	Add (T2, 30) to ATT, Add (P2, 30) to DPT
40	T1 changes status to “C(committed)” from “U(uncommitted)”
50	Add (T3, 50) to ATT
60	T1 removed from ATT
70	Update (T2, 70) on ATT, Add (P3, 70) to DPT
80	Update (T2, 80) on ATT

From the contents of this table we can easily construct the other ones, by the end of the Analysis phase, we have the ATT and DPT complete, as follows:

Active Transaction Table	
Transaction	Last LSN
T2	80
T3	50

Dirty Page Table	
Dirty Page	Prev LSN
P1	20
P2	30
P3	70

After the Redo phase, we have the following Redo table (starting at LSN = 20 as it’s the lower value in the DPT):

LSN	Action
20	Redo P1
30	Redo P2
40	-
50	Redo P2
60	-
70	Redo P3
80	-

Finally, after the Undo phase, we have the following Undo table (Initial ToUndo = [80, 50], from ATT):

LSN	Action
80	ToUndo = [70, 50]
70	Undo Update on P3 Write CRL to Log ToUndo = [50, 30]
50	Undo Update on P2 Write CRL to Log Write END to Log ToUndo = [30]
30	Undo Update on P2 Write CRL to Log Write END to Log ToUndo = []

And finally, the Log:

LSN	Prev LSN	XID	Type	Page ID	Undo Next
90	50	T3	CRL	P2	null
100	80	T2	CRL	P3	30
110	90	T3	END	-	-
120	100	T2	CRL	P2	null
130	120	T2	END	-	-

5.d

When using the read uncommitted isolation level, many anomalies may occur, like dirty reads, non-repeatable reads or phantom reads. To better understand how these may occur, we will start by giving a description of each one.

A dirty read occurs when a transaction reads an item that has been changed by another transaction that hasn't yet been committed.

A non-repeatable read can happen when a query in a transaction returns data that would be different if the same query were to be repeated within that same transaction. This happens because other transactions can modify the items retrieved and commit their actions.

A phantom read might occur when during a transaction, two identical queries are executed, and the collection of items returned by the second query is different from the first one, thus giving a different result as before in the same transaction. This is more usually considered when querying for counts, for example.

5.d.1

In this specific case all of these may occur, like a dirty read, that can occur when the manager is reading an entry for a given id, that is being updated simultaneously, as show in this example (using pseudo instructions for simplicity), and this can represent, e.g., the manager reading an entry being updated by the health professional, who then cancels the operation:

Health Professional	Manager
BEGIN TRANSACTION	
update schedule with id = 10	
	BEGIN TRANSACTION
	read schedule with id = 10 **(This instruction represents the dirty read)**
ROLLBACK TRANSACTION	
	read same schedule with id = 10 (gives different result within the same transaction)
	COMMIT TRANSACTION

In a similar fashion, a non-repeatable read might occur, if two similar reads are done within the same transaction by the manager and there is an update by a professional on the table, just like in the following example. This can represent the case in which a health professional is fixing a wrong entry on his/her schedule:

Health Professional	Manager
	BEGIN TRANSACTION
	read schedule with id = 10 (gives a result)
BEGIN TRANSACTION	
update schedule with id = 10	
COMMIT TRANSACTION	
	read same schedule with id = 10 (gives a different result from the one before) **(Cannot repeat the read from before anymore)**
	COMMIT TRANSACTION

5.d.2

A phantom read would happen in a similar way to a non-repeatable read, but if there was an insertion instead of an update. This can cause the manager to get an incorrect count of a certain health professional's entries in the table, in this case, Catarina's friend. For this example we will consider the friend's Id to be 2:

Nurse Catarina	Manager

	SET TRANSACTION ISOLATION LEVEL
	READ UNCOMMITTED;
	
	BEGIN TRANSACTION;
	
	SELECT COUNT(*)
	FROM Schedule
	WHERE professionalId = 2;
	(returns the current count)

BEGIN TRANSACTION;
 	
INSERT INTO Schedule
(scheduleId,professionalId,slotStart,slotEnd)
VALUES
(11,2,2021-06-03 00:00:00,2021-06-03 08:00:00);
 	
COMMIT TRANSACTION;	

	SELECT COUNT(*)
	FROM Schedule
	WHERE professionalId = 2;
	(returns a different count)
	(The count is considering a non-committed value)
	
	COMMIT TRANSACTION;

6 Database Tuning

6.a

```
SELECT name
FROM HealthProfessional;
```

No Index required for this query, since it will always require a full table scan.

```
SELECT professionalId, slotStart, slotEnd
FROM Schedule NATURAL JOIN HealthProfessional
WHERE role='nurse';
```

Two indexes can be used to improve this query execution time:

1. Index on HealthProfessional(Role), B+ Tree, Clustered. This will facilitate when filtering the HealthProfessional table entries, as well as speeding up the retrieval of the necessary data, keeping them physically close to each other.
2. Index on Schedule(professionalId), Hash, Non-Clustered to accelerate the join process between the two tables.

6.b

The presented subquery is a correlated subquery with aggregates.

The subquery is correlated since it uses values from the outer query and is aggregate since it uses the avg function, which aggregates values of several rows.

Independently of the query being correlated or uncorrelated, subqueries have a huge impact on performance because they are processed for each retrieved element.

The query could be rewritten as follows:

```
INSERT INTO temp
SELECT AVG(quantity) AS avg, type
        FROM stock
        GROUP BY type;


SELECT stockId
FROM stock NATURAL JOIN temp
        WHERE quantity = avg;
```

6.c

The query correspondent to the manager request is:

```
SELECT name, performanceEvaluation, slotStart, slotEnd
FROM HealthProfessional NATURAL JOIN Schedule
        WHERE role=nurse;
```

Denormalizing the table Schedule is a solution, where the name, performanceEvaluation and role attributes are added to Schedule:

Schedule(scheduleId, professionalId, slotStart, slotEnd, name, performanceEvaluation, role)

Additionally, an horizontal Partitioning of the Schedule Table by Role can be made to further improve performance:

Schedule_NURSE(scheduleId, professionalId, slotStart, slotEnd, name, performanceEvaluation)

Schedule_OTHER_ROLES(scheduleId, professionalId, slotStart, slotEnd, name, performanceEvaluation)

7 Miscellaneous

7.a

POP stands for Progressive Query Optimization and is a compromise between static and continuous optimization.
The main idea is to trigger an optimization when the cardinality estimation error indicates that the current query execution plan is suboptimal.
The reoptimization can happen in the middle of a query, so alternating execution and optimization steps can happen during the execution of a query multiple times.
During an execution step, POP monitors the real values and gives them to the next optimization step. POP triggers the optimization by inserting CHECKs in the query execution. Checkpoints (CHECK) are the POP points of control, which validate the optimizer's cardinality estimates against actual cardinalities.
Each CHECK has a check range and will suspend the execution if the number of rows violates the CHECK condition, this is, rows are counted and compared to the CHECK threshold.
Logic is used to determine the CHECK range, the CHECK locations and to exploit intermediate results to obtain optimizations.

7.b

Traditional projection is defined as taking a vertical subset from the columns of a single table and retaining the unique rows. This kind of projection returns some of the columns and all the rows in a table. These operations are row-oriented.

In the presented paper, a different concept of projections is presented. Data is saved in projections, which are groups of columns sorted on the same attribute. C-stores then store values by column and not by record. The logical tables are not stored but, instead, there are several projections and each contains a subset of attributes of one logical table and can also contain attributes from other tables as long as there is a foreign key from the original table to the other table.

For example, to perform a select on a table, instead of taking a subset of the columns of each row, column-oriented optimizer and executor are used, with different primitives than in a row-oriented system, for instance using storage keys associated with the segments of the projections, which allow to identify the records the columns are associated to.

7.c

Dataflow programming languages isolate local behaviors in actors, which consume data tokens on their inputs and produce new data on their outputs and are supposed to run in parallel and exchange data through point-to-point channels. There is no notion of central memory (both for code and data) unlike the Von Neumann model of computers.

This makes it easier for programmers to understand and control how their data processing task is executed.

For example, writing a Pig Latin program is similar to specifying a query execution plan (i.e., a dataflow graph).

To define reference models for video encoding, a traditional programming model makes the assumption that everything run as a sequence of operations, which is not useful and does not take advantage of the parallelism presented in computers' hardware (pipeline, VLIW, mutlicores, VLSI).

7.d

Ensuring atomicity, means to prevent updates to the database of occurring only partially, i.e., a transaction has either finished its execution and has saved all the changes it made to the database system or incomplete executions are completely undone. Durability is the property which guarantees that transactions that have committed will survive permanently

The recovery system ensures atomicity and durability by combining steal and no-force strategies with logs and checkpoints. The checkpoints will periodically write the data to disk to guarantee persistence, while the log will enable to check what was already written to disk and what wasn't due to a possible failure. The checkpoint operations consist on writing the current buffers to disk and updating the log files, allowing the desired properties to be achieved.

Durability is achieved by flushing the transaction's log records to non-volatile storage before acknowledging commitment, this is, by saving all the changes made to the system in a log and whenever a crash occurs all the committed transactions are maintained.

When a system crash occurs recovery manager achieves atomicity by undoing all the actions or changes done to a system that did not commit and redoing committed ones.

7.e

One heuristic rule that allows to reduce the number of execution plans generated for a given query is to apply SELECT and PROJECT operations before applying the JOIN.
The size of the resulting table from an operation such as a JOIN is, in most of the cases, a multiplicative function of the sizes of the input tables. The SELECT and PROJECT operations reduce the size of the resulting table, therefore should be applied before a join or other similar operation.