
LUT
University
LAND
OF THE
CURIOUS

CT60A4304 - BASICS OF DATABASE SYSTEMS

INDEX AND OPTIMIZATION

Lecture

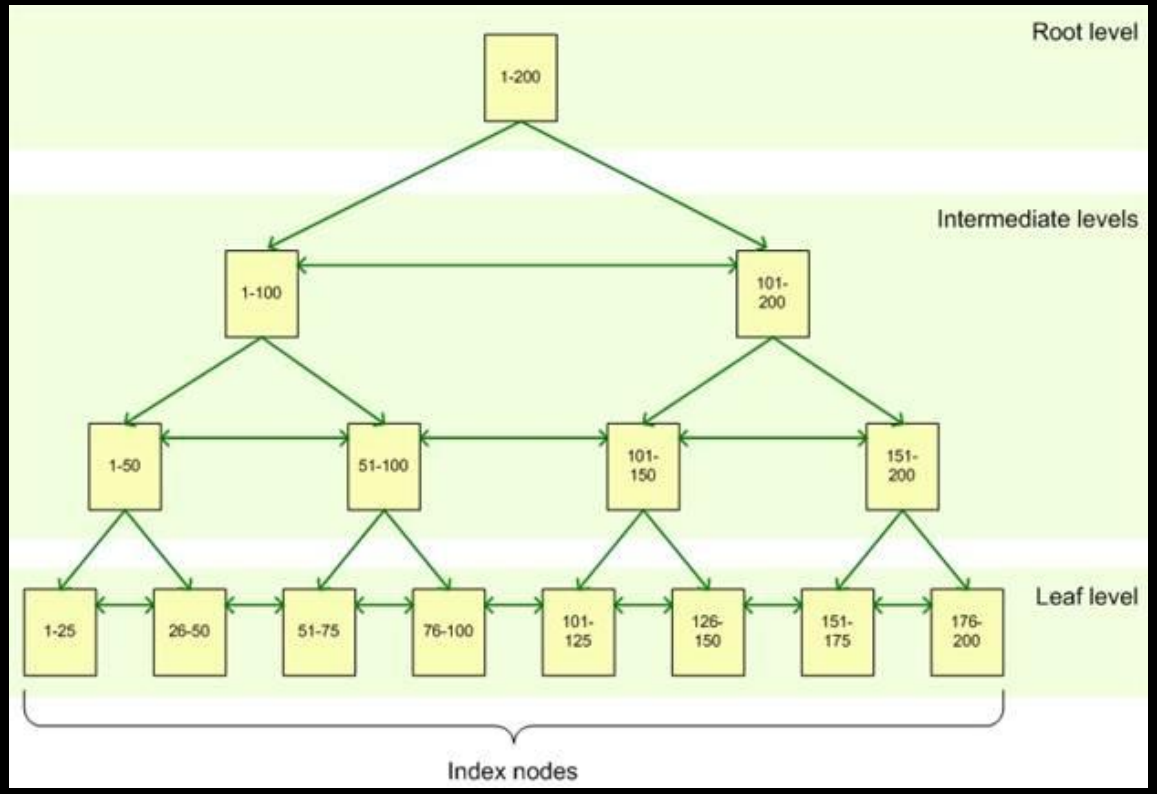
Jiri Musto, D.Sc.

TABLE OF CONTENTS

INDICES / INDEXES IN GENERAL

- Dictionary style data structure
- Purpose is to have faster queries
- No need to go through all rows, indices point to the “correct” rows and enables faster retrieval
- How indices work
  - B-tree (balanced tree) algorithm
  - Speeds up SELECT queries with WHERE statements
  - Makes UPDATE and INSERT slower
  - Can be created or dropped with no effect on data

HOW INDICES WORK



https://www.red-gate.com/simple-talk/databases/sql-server/learn/sql-server-index-basics/

WHEN TO USE INDICES

- Columns are often used in search conditions (WHERE or JOIN statements)
- Columns are often used in ordering results
- Column has a wide range of values
- Column has lots of null values
- Table is large and most queries retrieve 2 to 4 % of rows

WHEN NOT TO USE INDICES

- Table is small
- Most queries retrieve more than 2 to 4 % of rows
- Table is often updated
  - Indices can be removed temporarily when performing updates
- Columns are rarely used for queries

- These are basic guidelines, you should test what works for you

INDICES IN SQL

- CREATE INDEX index_name ON table_name(columns);
- The order of columns is important!
  - In most databases, columns are used in-order
- DROP INDEX index_name

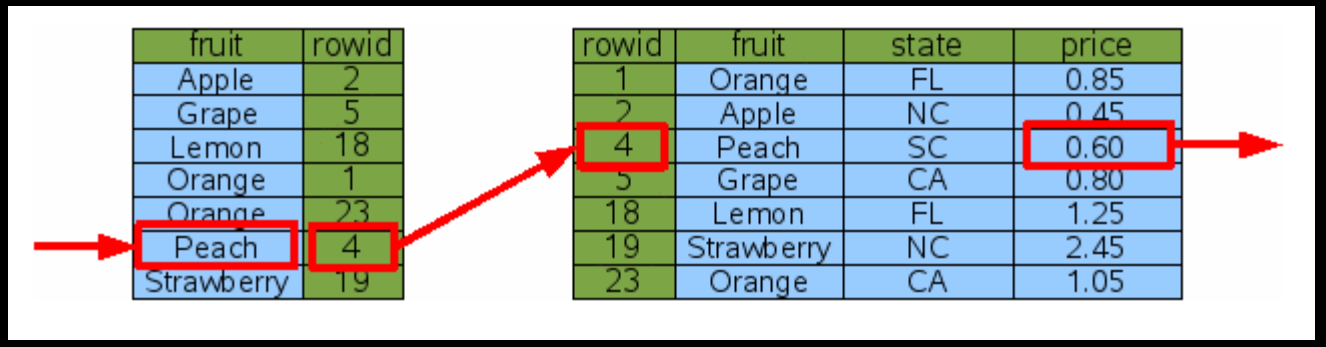
```
 9  CREATE INDEX PlayerIndex ON Player(last_name);
10  CREATE INDEX RankingIndex ON Player(rank, score);
11
```

EXAMPLE: HOW INDICES WORK

SELECT price FROM fruitsforsale WHERE fruit='Peach';

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Without index



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

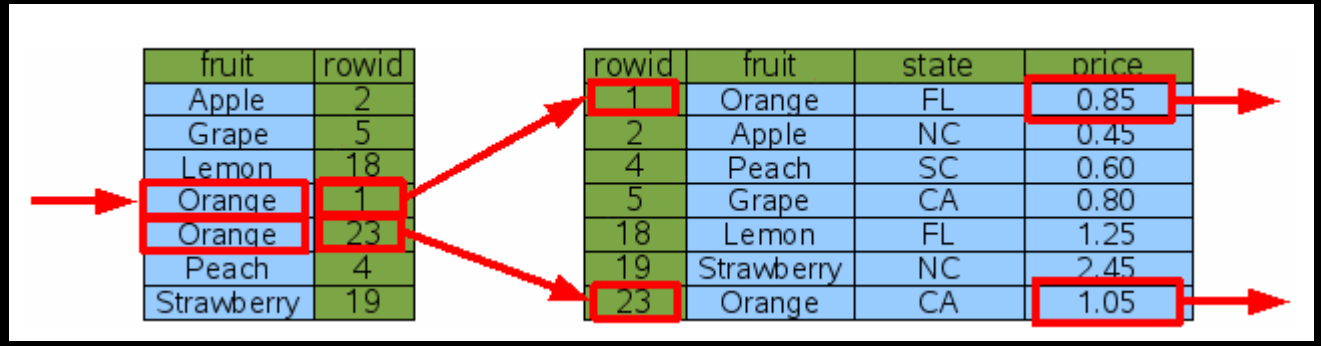
rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

With index

https://www.sqlite.org/queryplanner.html

EXAMPLE: HOW INDICES WORK

SELECT price FROM fruitsforsale WHERE fruit='Orange'



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

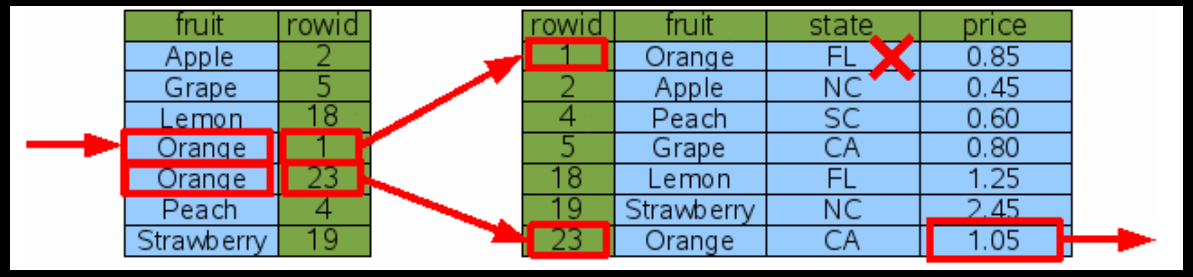
rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

https://www.sqlite.org/queryplanner.html

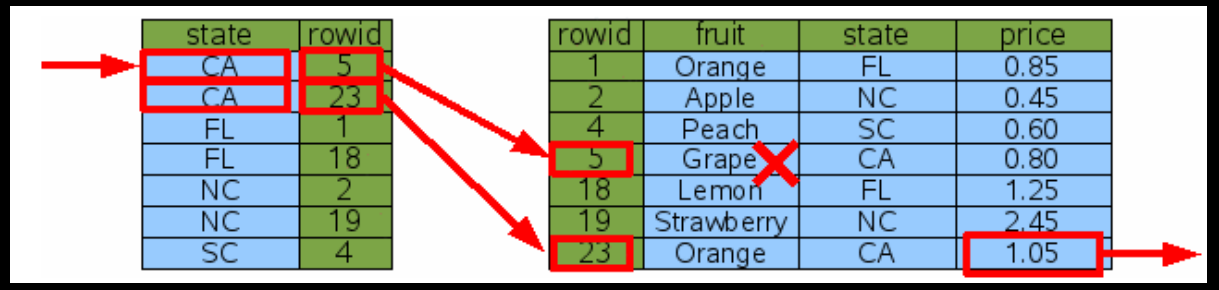
EXAMPLE: HOW INDICES WORK

SELECT price FROM fruitsforsale WHERE fruit='Orange' AND state='CA'

Using fruit index



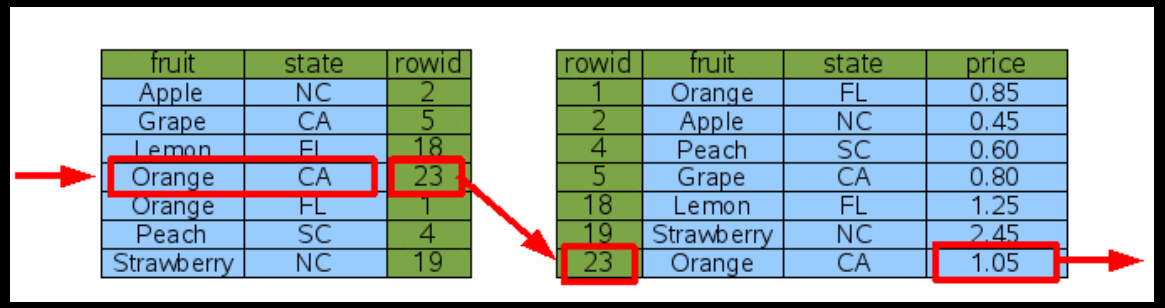
Using state index



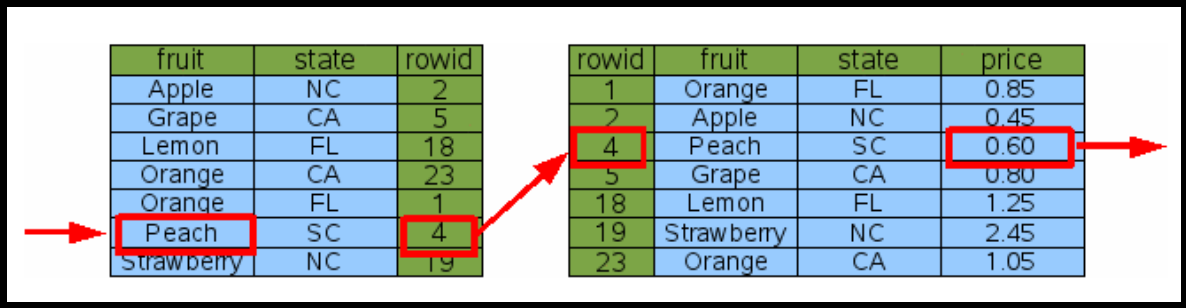
https://www.sqlite.org/queryplanner.html

EXAMPLE: MULTI-COLUMN INDEX

SELECT price FROM fruitsforsale WHERE fruit='Orange' AND state='CA'



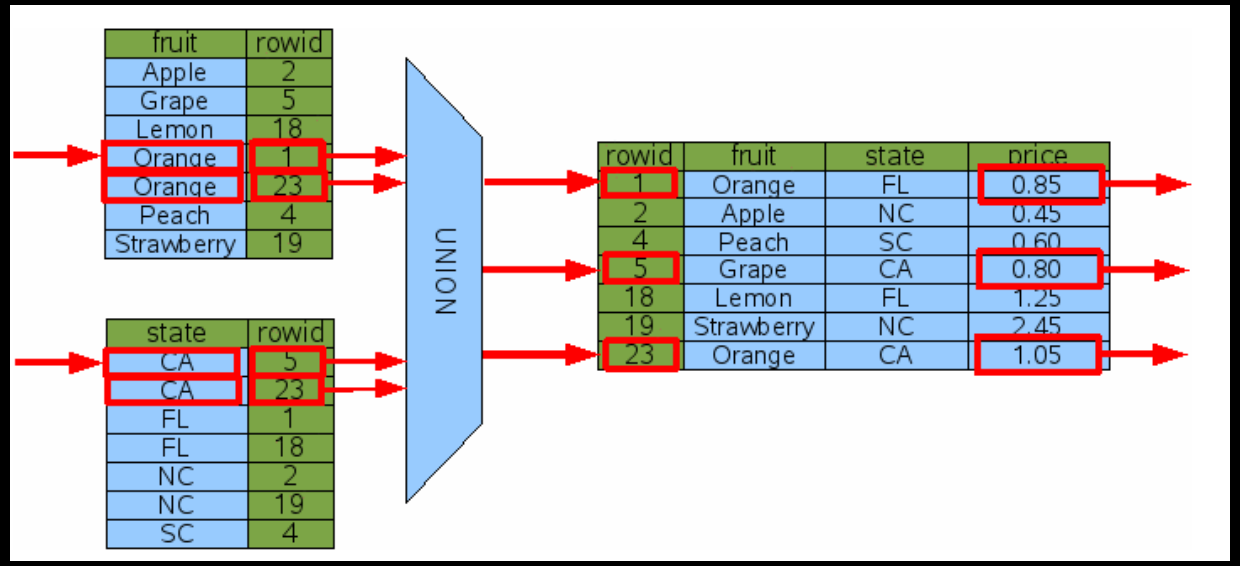
SELECT price FROM fruitsforsale WHERE fruit='Peach'



https://www.sqlite.org/queryplanner.html

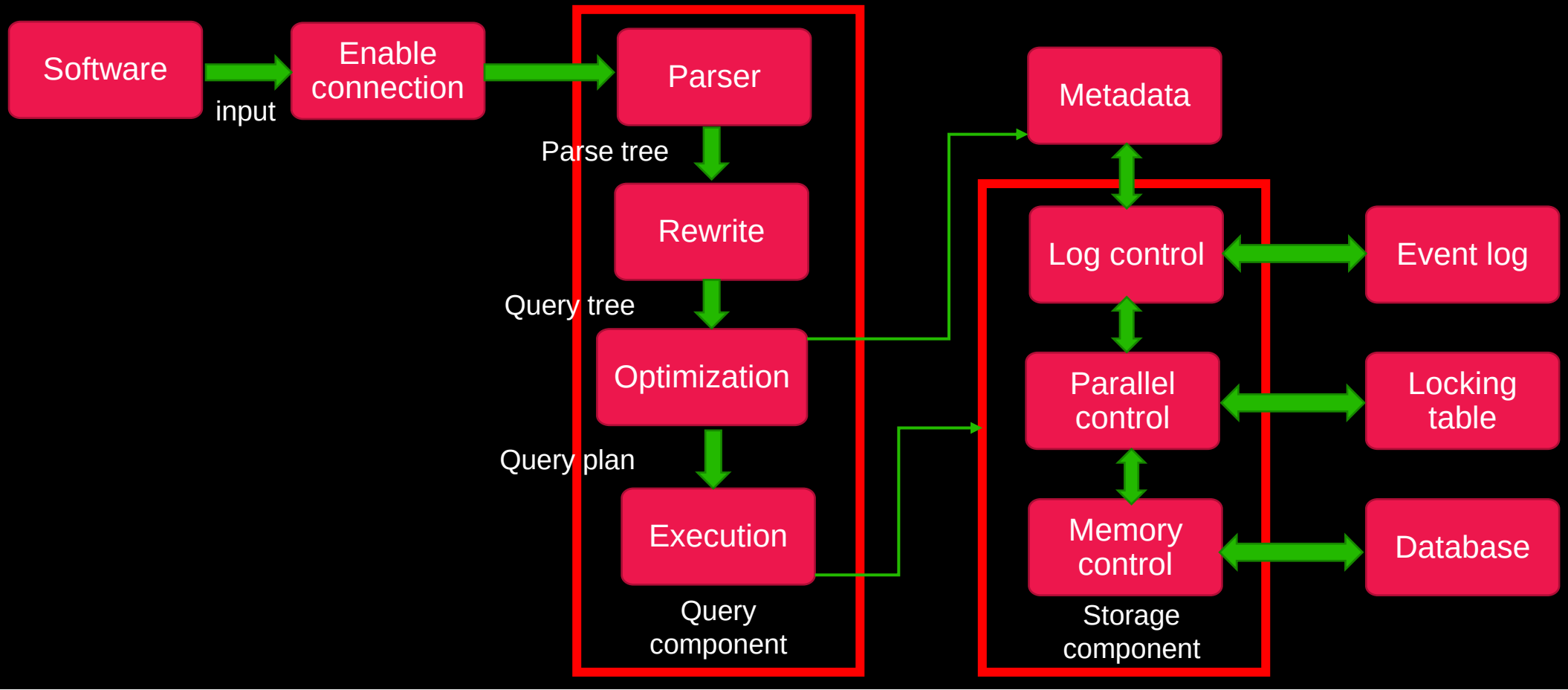
EXAMPLE: OR CONDITION

SELECT price FROM fruitsforsale WHERE fruit='Orange' OR state='CA'



https://www.sqlite.org/queryplanner.html

QUERY PROCESSING PHASES



QUERY PLANNER

- Database optimizes the query and makes a query plan based on the SQL query and database structure
- Each DBMS has commands for timers and viewing query plans
  - SQLite, PostgreSQL and MySQL has EXPLAIN (QUERY PLAN)
- SQLite timer is: .timer on
- Shows three different times: real, user, sys

```
Run Time: real 0.014 user 0.000000 sys 0.000000
```

  - real time is the elapsed time
  - user time is the time spent executing instructions in user mode
  - sys time is the time spent executing instructions in supervisor mode

EXPLAIN (QUERY PLAN)

- A high-level description of the strategy to implement a specific query

```
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     54    0                    0   Start at 54
1     OpenRead       1     5     0     5              0   root=5 iDb=0; Ranking
2     OpenRead       0     2     0     5              0   root=2 iDb=0; Player
3     Rewind         1     53    0                    0
4       Column         1     4     1                    0   r[1]=Ranking.FK_playerid
5       SeekRowid      0     52    1                    0   intkey=r[1]
6       Integer        23    4     0                    0   r[4]=23; return address
7       Null           0     5     5                    0   r[5..5]=NULL; Init subquery result
8       Integer        1     6     0                    0   r[6]=1; LIMIT counter
9       Null           0     7     7                    0   r[7..7]=NULL
10      OpenRead       2     4     0     3              0   root=4 iDb=0; Matches
11      Rewind         2     20    0                    0
12        Rowid          0     8     0                    0   r[8]=rowid
13        Column         2     1     9                    0   r[9]=Matches.FK_playerOne
14        Eq             9     18    8     BINARY-8       67  if r[8]==r[9] goto 18
15        Rowid          0     9     0                    0   r[9]=rowid
16        Column         2     2     8                    0   r[8]=Matches.FK_playerTwo
17        Ne             8     19    9     BINARY-8       83  if r[9]!=r[8] goto 19
18        AggStep        0     0     7     count(0)       0   accum=r[7] step(r[0])
19      Next           2     12    0                    1
20      AggFinal       7     0     0     count(0)       0   accum=r[7] N=0
21      Copy           7     5     0                    0   r[5]=r[7]
22      DecrJumpZero   6     23    0                    0   if (--r[6])==0 goto 23
```

```
QUERY PLAN
|--SCAN Ranking
|--SEARCH Player USING INTEGER PRIMARY KEY (rowid=?)
|--CORRELATED SCALAR SUBQUERY 1
|  `--SCAN Matches
`--CORRELATED SCALAR SUBQUERY 2
   `--SCAN Matches
sqlite>
```

EXAMPLE OF QUERY PLANS

```
QUERY PLAN
|--SCAN M1
|--MULTI-INDEX OR
|   |--INDEX 1
|   |   `--SEARCH P1 USING INTEGER PRIMARY KEY (rowid=?)
|   `--INDEX 2
|       `--SEARCH P1 USING INTEGER PRIMARY KEY (rowid=?)
|--MULTI-INDEX OR
|   |--INDEX 1
|   |   `--SEARCH P2 USING INTEGER PRIMARY KEY (rowid=?)
|   `--INDEX 2
|       `--SEARCH P2 USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH R1 USING AUTOMATIC COVERING INDEX (FK_playerid=?)
|--SEARCH R2 USING AUTOMATIC COVERING INDEX (FK_playerid=?)
`--USE TEMP B-TREE FOR ORDER BY
```

```
QUERY PLAN
|--MATERIALIZE SUBQUERY 2
|   `--COMPOUND QUERY
|       |--LEFT-MOST SUBQUERY
|       |   `--SCAN Matches
|       `--UNION USING TEMP B-TREE
|           `--SCAN Matches
|--SCAN LoserRanking
|--SEARCH SUBQUERY 2 USING AUTOMATIC COVERING INDEX (LoserID=?)
|--SEARCH Winner USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH Loser USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH WinnerRanking USING AUTOMATIC COVERING INDEX (FK_playerid=?)
`--USE TEMP B-TREE FOR ORDER BY
```

```
QUERY PLAN
|--SCAN M
|--SEARCH W USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH Ws USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH L USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH Ls USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR ORDER BY
```

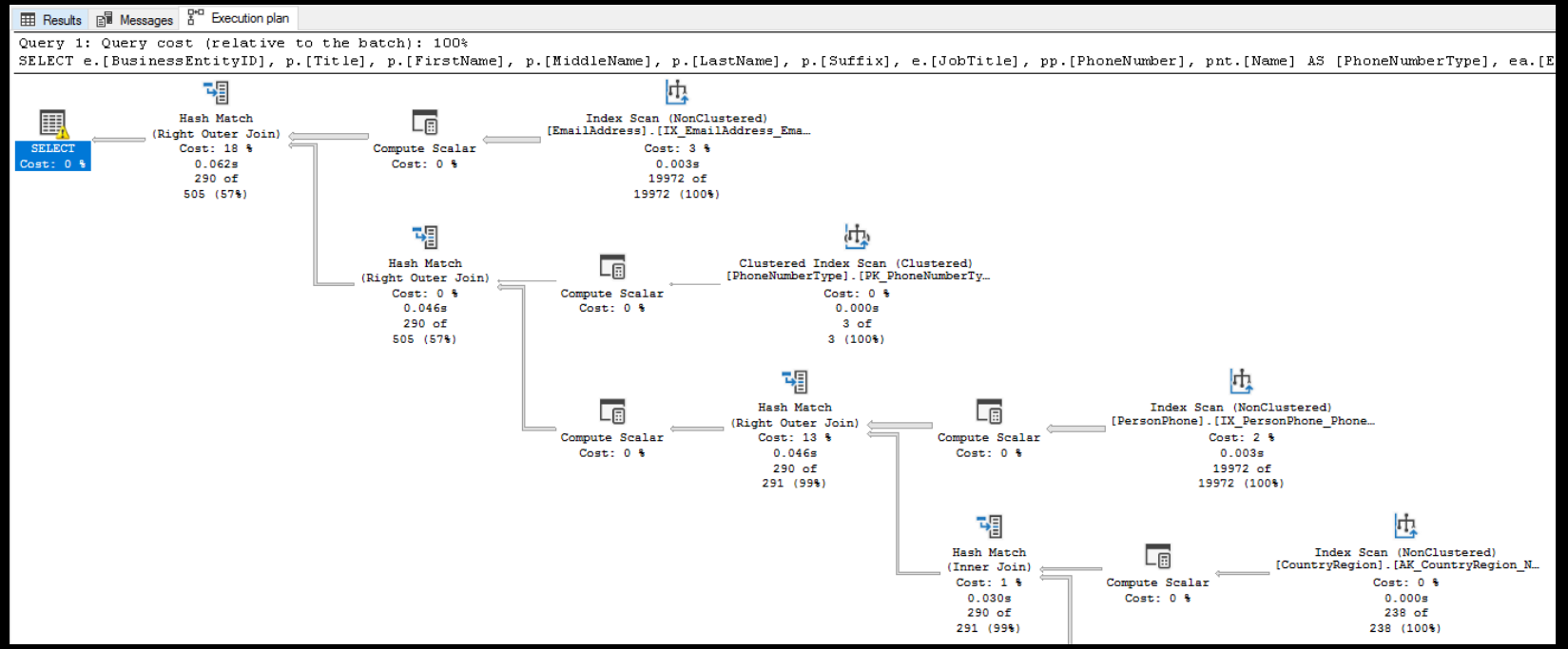
QUERY EXPLANATION IN OTHER DBMS

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

Postgres

QUERY EXPLANATION IN OTHER DBMS



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	403
Actual Number of Rows	3
Actual Number of Batches	0
Estimated I/O Cost	0.0120139
Estimated Operator Cost	0.0126142 (52%)
Estimated CPU Cost	0.0006003
Estimated Subtree Cost	0.0126142
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	3
Estimated Number of Rows to be Read	403
Estimated Row Size	409 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Predicate
[WideWorldImportersDW].[Dimension].[Customer].[Postal Code] =N'90761'

Object
[WideWorldImportersDW].[Dimension].[Customer].[PK_Dimension_Customer]

Output List
[WideWorldImportersDW].[Dimension].[Customer].Customer Key,

Execution plan in SQL Server

JOIN VS SUBQUERY

- JOIN is generally faster than subquery
- NOT because JOIN is inherently faster
- ...But because automatic query optimizer is better at handling JOINs. Subqueries need more manual optimization
- There are two types of subqueries:
  - Correlated subqueries: Subquery that is run on each row, often inside the outer SELECT statement
  - Nested subqueries: Subquery is run separately, included in WHERE, FROM or JOIN condition
- Both are valid options
  - JOINs are easier to use in general but sometimes a subquery may speed up the process if done correctly
  - Sometimes a subquery cannot be rewritten as a JOIN, thus subquery is the better (only) option

OPTIMIZING QUERIES

- Less data you retrieve, the better
  - Limit the amount of tables, rows and columns
  - You can limit results for sampling
- Use the minimum amount of queries so optimizer can work its magic
  - But remember, optimizer is not perfect
- Use indices to speed up the most important parts of queries
  - Over doing indices may slow the queries
- Do not store large amounts of binary data in a database (use a separate file)
- Mass insert is always faster than multiple inserts
  - One insert adding 1000 rows vs. 1000 inserts adding one row
- There are useful functions, such as conversion, substrings and dateparts for query conditions but these functions slow down the query

OPTIMIZING QUERIES

- JOIN is better than WHERE condition
- WHERE is faster than HAVING (when possible)
- EXISTS is faster than COUNT() or IN
- IN condition is slow compared to “ < AND > “
  - NOT queries are also slower than regular ones
- Limit the usage of DISTINCT when possible
- Comparing different data types in SQL queries requires the conversion of one type to match another
  - Implicit conversion is done for the whole table before the query is executed


LUT
University