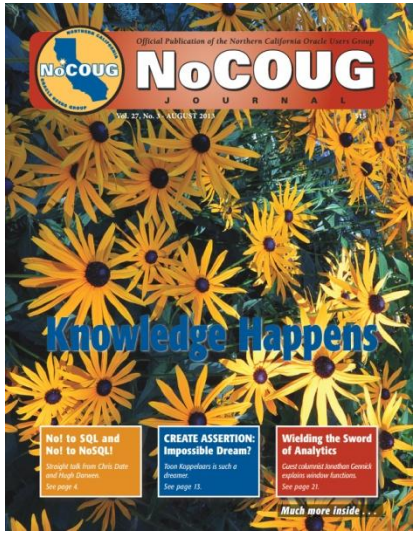
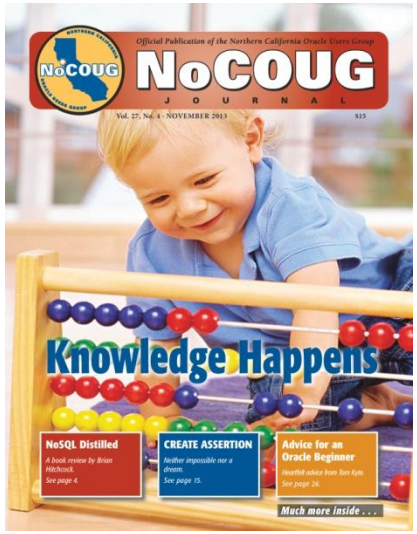
12 Things Every Oracle DBA and Developer Should Know About SQL

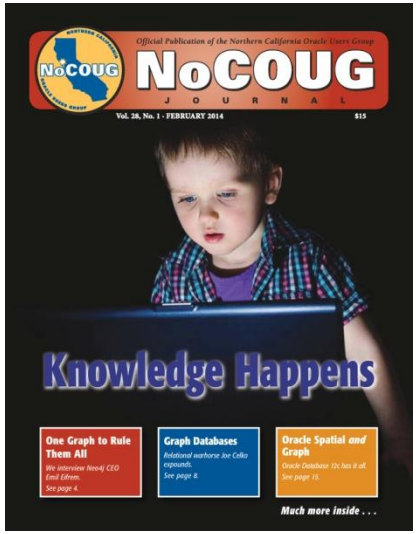
Iggy Fernandez

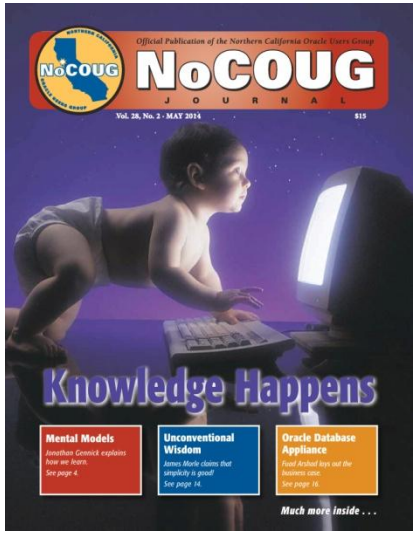
Introduction

- Editor of the *NoCOUG Journal* (Archive)
- Presentation based on "The Twelve Days of SQL" series of blog posts
- Also read "The Twelve Days of NoSQL" series









Top Tips

- Buy every book by Steven Faroult
  - The Art of SQL
  - Refactoring SQL Applications
  - SQL Success
- Follow my ToadWorld blog
  - Hitchhiker's Guide to the EXPLAIN PLAN
- Install the OTN Developer Day virtual machine on your desktop.

“Those who are in love with practice without knowledge are like the sailor who gets into a ship without rudder or compass and who never can be certain where he is going. Practice must always be founded on sound theory.”

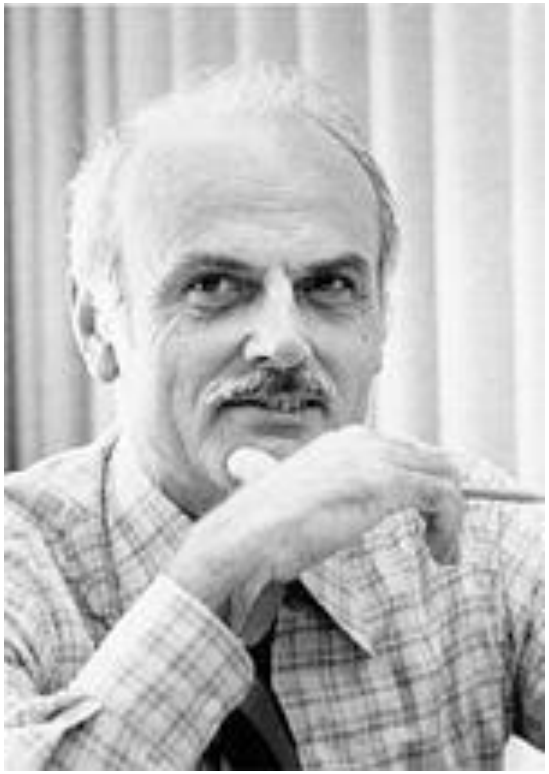
—The Discourse on Painting

by Leonardo da Vinci

#1 SQL IS A NON-PROCEDURAL LANGUAGE

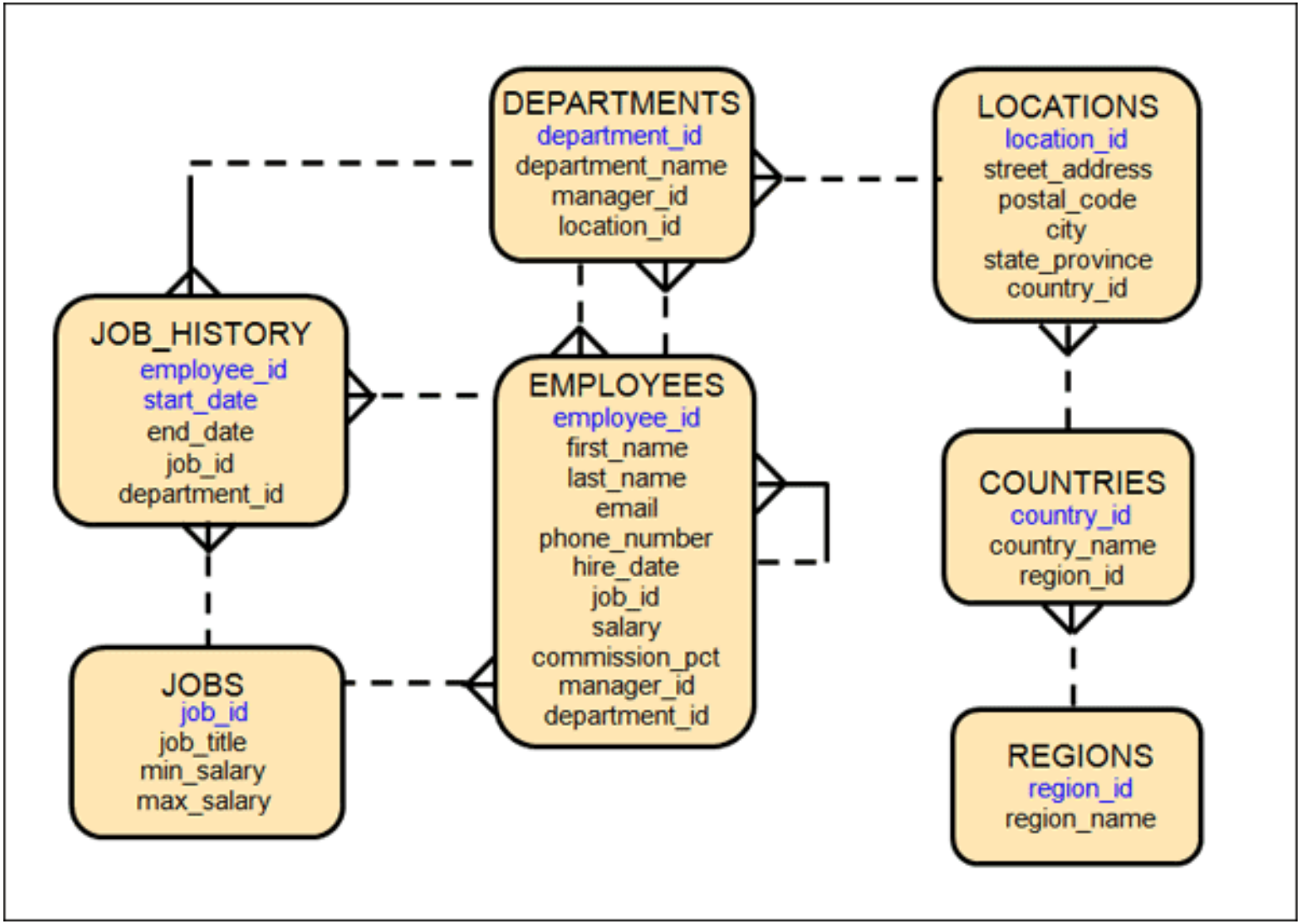
Patron Saints of Databases

Charles Bachman
1973 Turing Award Winner

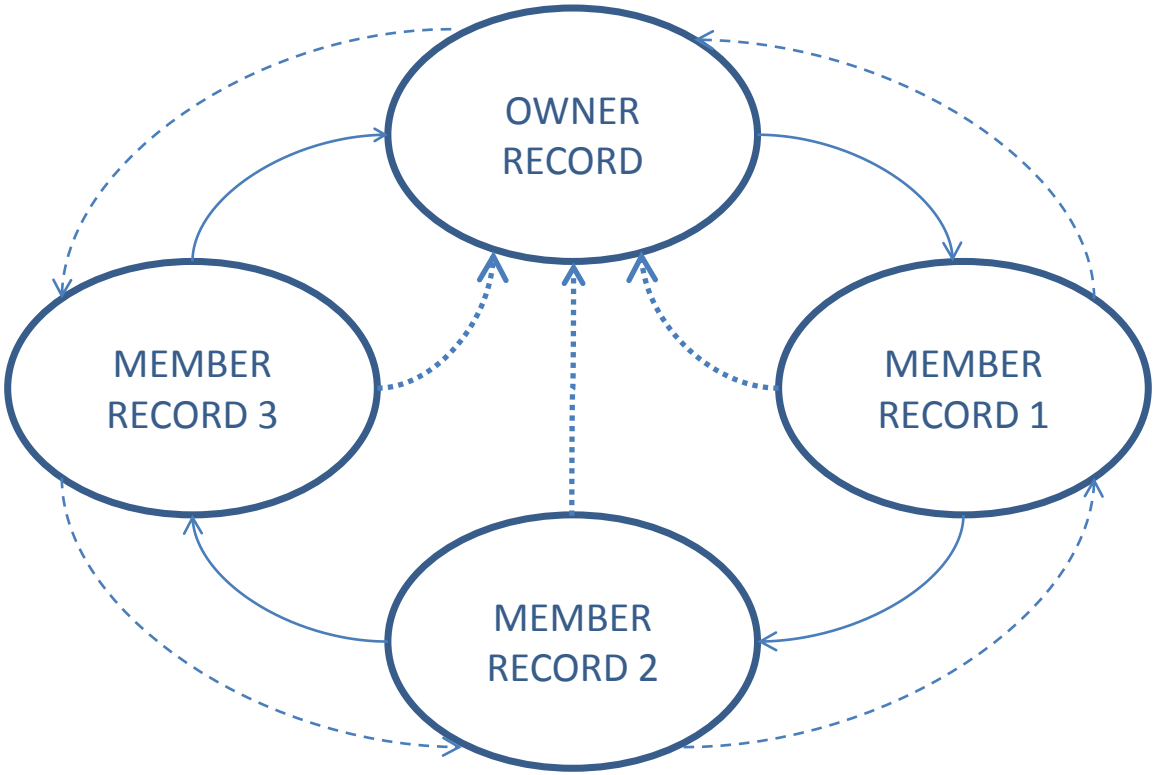


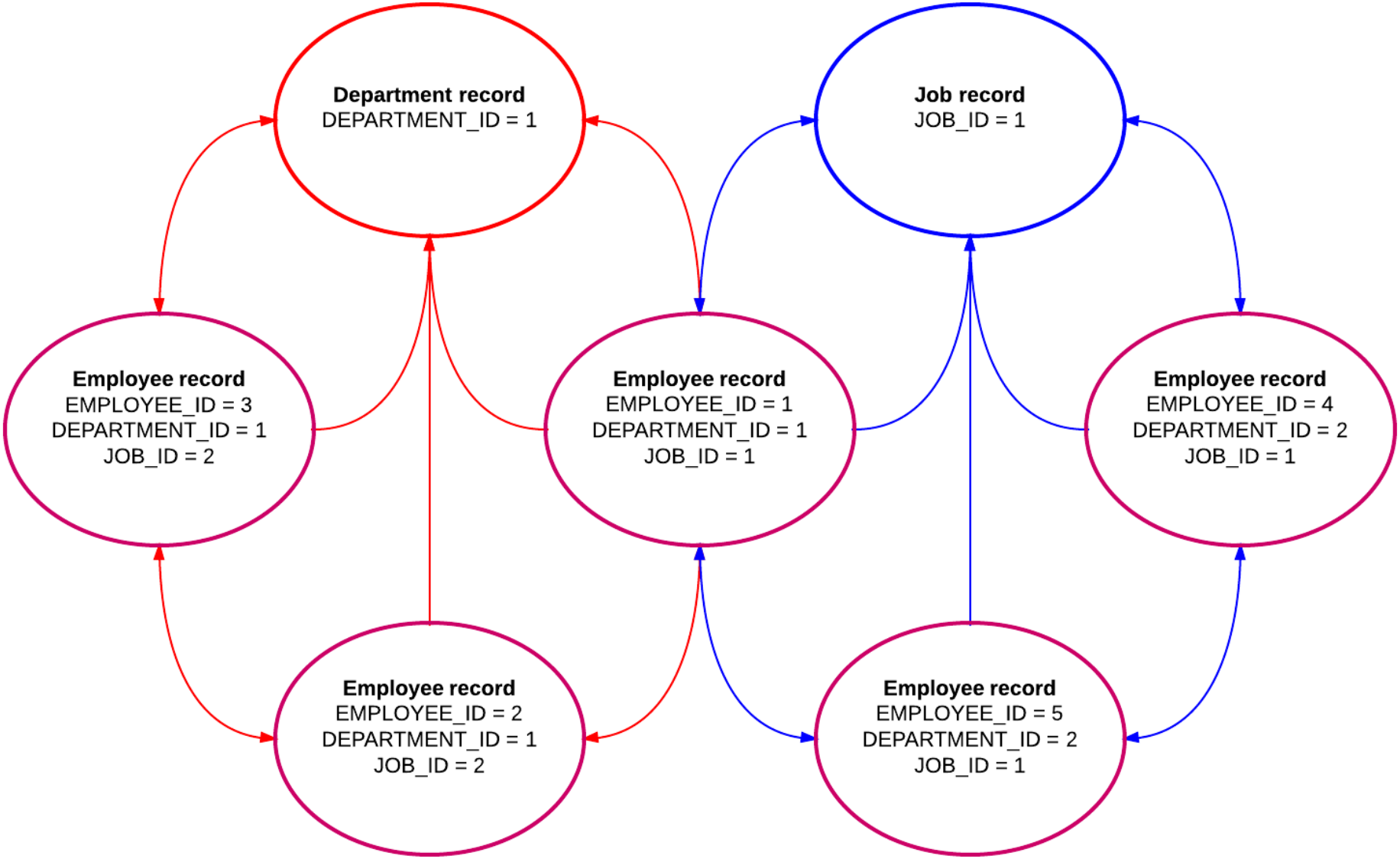
Dr. Edgar Codd
1981 Turing Award Winner

HR Schema


DEPARTMENTS
department_id
department_name
manager_id
location_id
LOCATIONS
location_id
street_address
postal_code
city
state_province
country_id
JOB_HISTORY
employee_id
start_date
end_date
job_id
department_id
EMPLOYEES
employee_id
first_name
last_name
email
phone_number
hire_date
job_id
salary
commission_pct
manager_id
department_id
COUNTRIES
country_id
country_name
region_id
JOBS
job_id
job_title
min_salary
max_salary
REGIONS
region_id
region_name

Network Model Chains




Department record
DEPARTMENT_ID = 1
Job record
JOB_ID = 1
Employee record
EMPLOYEE_ID = 3
DEPARTMENT_ID = 1
JOB_ID = 2
Employee record
EMPLOYEE_ID = 1
DEPARTMENT_ID = 1
JOB_ID = 1
Employee record
EMPLOYEE_ID = 4
DEPARTMENT_ID = 2
JOB_ID = 1
Employee record
EMPLOYEE_ID = 2
DEPARTMENT_ID = 1
JOB_ID = 2
Employee record
EMPLOYEE_ID = 5
DEPARTMENT_ID = 2
JOB_ID = 1

The Programmer as Navigator

1. Sequential access
2. ROWID access
3. Primary key access
4. Secondary key access
5. Starting from the owner record, get all the records in a chain
6. Starting from any record, get the prior and next in a chain
7. Starting from any record, get the owner record

“Each of these access methods is interesting in itself, and all are very useful. However, it is the synergistic usage of the entire collection which gives the programmer great and expanded powers to come and go within a large database while accessing only those records of interest in responding to inquiries and updating the database in anticipation of future inquiries.”

—Charles Bachman

ACM Turing Award Lecture, 1973

Relational Model

"In the choice of logical data structures that a system is to support, there is one consideration of absolutely paramount importance – and that is the ***convenience of the majority of users****. … To make formatted data bases readily accessible to users (****especially casual users****) who have* ***little or no training in programming*** *we must provide the simplest possible data structures and almost natural language. … What could be a simpler, more universally needed, and more universally understood data structure than a table?"*

—Dr. Edgar Codd

Normalized Data Base Structure: A Brief Tutorial (1971)

The intended audience for SQL

"There is a large class of users who, while they are not computer specialists, would be willing to learn to interact with a computer in a reasonably high-level, non-procedural query language. Examples of such users are ***accountants, engineers, architects, and urban planners****. It is for this class of users that SEQUEL is intended."*

—Donald Chamberlin and Raymond Boyce

Sequel: A Structured English Query Language (1974)

What can you expect from a non-procedural query language?

- The query runs slowly.
- The query ran fast yesterday but is running slowly today.
- The query ran fast a few minutes ago but is running slowly now.
- The query runs fast in the QA database but runs slowly in the production database.
- The query runs fast in other production databases but not in this production database.
- Oracle is not using the indexes we created.
- The query plan is not the one we expected.
- **The query runs slower after a database upgrade.**

Query Plan Stability

- Hints
- Stored Outlines
- SQL Plan Management (Oracle Database 12c Enterprise Edition only)
- Don’t refresh statistics?
- Don’t use bind variable peeking?

#2 SQL IS BASED ON RELATIONAL CALCULUS AND RELATIONAL ALGEBRA

Definition of a relational database

"A relational database is a database in which: The data is perceived by the user as tables (and nothing but tables) and the operators available to the user for (for example) retrieval are operators that derive "new" tables from "old" ones."

—Chris Date

An Introduction to Database Systems

Relational Operators (Fundamental and sufficient)

- Selection
- Projection
- Union
- Difference
- Join (Cartesian Join)

The Employees table

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4

The Job History table

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

The Jobs table

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

Employees who have held all accounting jobs (job_id like 'AC%')

```
WITH
  -- Step 1: Projection
  all_employees_1 AS
  ( SELECT employee_id FROM employees
  ),
  -- Step 2: Projection
  all_employees_2 AS
  ( SELECT employee_id FROM employees
  ),
  -- Step 3: Projection
  all_jobs AS
  ( SELECT job_id FROM jobs
  ),
```

```
  -- Step 4: Selection
  selected_jobs AS
  ( SELECT * FROM all_jobs WHERE job_id LIKE 'AC%'
  ),
  -- Step 5: Join
  selected_pairings AS
  ( SELECT * FROM all_employees_2 CROSS JOIN
selected_jobs
  ),
  -- Step 6: Projection
  current_job_titles AS
  ( SELECT employee_id, job_id FROM employees
  ),
```

```
-- Step 7: Projection
previous_job_titles AS
( SELECT employee_id, job_id FROM job_history
),
-- Step 8: Union
complete_job_history AS
( SELECT * FROM current_job_titles
UNION
SELECT * FROM previous_job_titles
),
-- Step 9: Difference
nonexistent_pairings AS
( SELECT * FROM selected_pairings
MINUS
SELECT * FROM complete_job_history
),
```

```
  -- Step 10: Projection
  ineligible_employees AS
  ( SELECT employee_id FROM nonexistent_pairings
  )
-- Step 11: Difference
SELECT * FROM all_employees_1
MINUS
SELECT * FROM ineligible_employees
```

Employees who have held all accounting jobs (Compact version)

```
SELECT employee_id FROM employees
MINUS
SELECT employee_id
FROM
  (SELECT employees.employee_id,
    jobs.job_id
  FROM employees
  CROSS JOIN jobs
  WHERE jobs.job_id LIKE 'AC%'
  MINUS
    ( SELECT employee_id, job_id FROM job_history
    UNION
    SELECT employee_id, job_id FROM employees
    )
  )
```

Employees who have held all accounting jobs (Relational calculus)

```
SELECT employee_id
FROM employees e
WHERE NOT EXISTS
  (SELECT job_id
  FROM jobs j
  WHERE job_id LIKE 'AC%'
  AND NOT EXISTS
    (SELECT *
    FROM
      ( SELECT employee_id, job_id FROM job_history
      UNION
      SELECT employee_id, job_id FROM employees
      )
    WHERE employee_id = e.employee_id
    AND job_id        = j.job_id
    )
  )
```

Additional Relational Operations (Derivable from the fundamental set)

- Intersection
- Natural join
- Equi-join
- Theta-join
- Left outer join
- Right outer join
- Full outer join
- Semi-join
- Anti-join
- Division

#3 THERE ISN'T ALWAYS A SINGLE OPTIMAL QUERY PLAN FOR A SQL QUERY

It depends on what the meaning of the word “is” is

```
select * from employees
where first_name like 'Lex'
and last_name like 'De Haan'

select * from employees
where first_name like :b1
and last_name like :b2
```

#4 THE TWELVE DAYS OF SQL: THE WAY YOU WRITE YOUR QUERY MATTERS

The Personnel table

```
CREATE TABLE personnel
(
  empid CHAR(9),
  lname CHAR(15),
  fname CHAR(12),
  address CHAR(20),
  city CHAR(20),
  state CHAR(2),
  ZIP CHAR(5)
);
```

The Payroll table

```
CREATE TABLE payroll
(
  empid CHAR(9),
  bonus INTEGER,
  salary INTEGER
);
```

Solution #1
Relational algebra method

```
SELECT DISTINCT per.empid, per.lname
FROM personnel per JOIN payroll pay ON (per.empid = pay.empid)
WHERE pay.salary = 199170;

Plan hash value: 3901981856
-----------------------------------------------------
| Id  | Operation                     | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |              |
|   1 |  HASH UNIQUE                  |              |
|   2 |   NESTED LOOPS                |              |
|   3 |    NESTED LOOPS               |              |
|   4 |     TABLE ACCESS BY INDEX ROWID| PAYROLL      |
|*  5 |      INDEX RANGE SCAN         | PAYROLL_I1   |
|*  6 |     INDEX UNIQUE SCAN         | PERSONNEL_U1 |
|   7 |    TABLE ACCESS BY INDEX ROWID | PERSONNEL    |
-----------------------------------------------------
```

Solution #2
Uncorrelated subquery

```
SELECT per.empid, per.lname
FROM personnel per
WHERE per.empid IN (SELECT pay.empid
FROM payroll pay
WHERE pay.salary = 199170);

Plan hash value: 3342999746
-----------------------------------------------------
| Id  | Operation                     | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |              |
|   1 |  NESTED LOOPS                 |              |
|   2 |   NESTED LOOPS                |              |
|   3 |    TABLE ACCESS BY INDEX ROWID| PAYROLL      |
|*  4 |     INDEX RANGE SCAN          | PAYROLL_I1   |
|*  5 |    INDEX UNIQUE SCAN          | PERSONNEL_U1 |
|   6 |   TABLE ACCESS BY INDEX ROWID | PERSONNEL    |
-----------------------------------------------------
```

Solution #3
Correlated subquery

```
SELECT per.empid, per.lname
FROM personnel per
WHERE EXISTS (SELECT *
  FROM payroll pay
  WHERE per.empid = pay.empid
  AND pay.salary = 199170);

Plan hash value: 864898783
-----------------------------------------------------
| Id  | Operation                     | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |              |
|   1 |  NESTED LOOPS                 |              |
|   2 |   NESTED LOOPS                |              |
|   3 |    SORT UNIQUE                |              |
|   4 |     TABLE ACCESS BY INDEX ROWID| PAYROLL      |
|*  5 |      INDEX RANGE SCAN         | PAYROLL_I1   |
|*  6 |    INDEX UNIQUE SCAN          | PERSONNEL_U1 |
|   7 |   TABLE ACCESS BY INDEX ROWID | PERSONNEL    |
-----------------------------------------------------
```

Solution #4
Scalar subquery in the WHERE clause

```
SELECT per.empid, per.lname
FROM personnel per
WHERE (SELECT pay.salary FROM payroll pay WHERE pay.empid = per.empid) = 199170;

Plan hash value: 3607962630
-------------------------------------------------
| Id  | Operation                    | Name       |
-------------------------------------------------
|   0 | SELECT STATEMENT             |            |
|*  1 |  FILTER                      |            |
|   2 |   TABLE ACCESS FULL          | PERSONNEL  |
|   3 |   TABLE ACCESS BY INDEX ROWID| PAYROLL    |
|*  4 |    INDEX UNIQUE SCAN         | PAYROLL_U1 |
-------------------------------------------------
   1 - filter(=199170)
   4 - access("PAY"."EMPID"=:B1)
```

Solution #5
Scalar subquery in the SELECT clause

```
SELECT DISTINCT pay.empid, (SELECT lname FROM personnel per WHERE per.empid = pay.empid)
FROM payroll pay
WHERE pay.salary = 199170;

Plan hash value: 750911849
-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|   1 |  TABLE ACCESS BY INDEX ROWID | PERSONNEL    |
|*  2 |   INDEX UNIQUE SCAN          | PERSONNEL_U1 |
|   3 |  HASH UNIQUE                 |              |
|   4 |   TABLE ACCESS BY INDEX ROWID| PAYROLL      |
|*  5 |    INDEX RANGE SCAN          | PAYROLL_I1   |
-----------------------------------------------------
   2 - access("PER"."EMPID"=:B1)
   5 - access("PAY"."SALARY"=199170)
```

Solution #6
Aggregate function to check existence

```
SELECT per.empid, per.lname
FROM personnel per
WHERE (SELECT count(*) FROM payroll pay WHERE pay.empid = per.empid AND pay.salary =
199170) > 0;

Plan hash value: 3561519015
-------------------------------------------------
| Id  | Operation                     | Name       |
-------------------------------------------------
|   0 | SELECT STATEMENT              |            |
|*  1 |  FILTER                       |            |
|   2 |   TABLE ACCESS FULL           | PERSONNEL  |
|   3 |   SORT AGGREGATE              |            |
|*  4 |    TABLE ACCESS BY INDEX ROWID| PAYROLL    |
|*  5 |     INDEX UNIQUE SCAN         | PAYROLL_U1 |
-------------------------------------------------
   1 - filter(>0)
   4 - filter("PAY"."SALARY"=199170)
   5 - access("PAY"."EMPID"=:B1)
```

Solution #7
Correlated subquery (double negative)

```
SELECT per.empid, per.lname
FROM personnel per
WHERE NOT EXISTS (SELECT *
 FROM payroll pay
 WHERE pay.empid = per.empid
 AND pay.salary != 199170);

Plan hash value: 103534934
-----------------------------------------
| Id  | Operation            | Name      |
-----------------------------------------
|   0 | SELECT STATEMENT     |           |
|*  1 |  HASH JOIN RIGHT ANTI|           |
|*  2 |   TABLE ACCESS FULL  | PAYROLL   |
|   3 |   TABLE ACCESS FULL  | PERSONNEL |
-----------------------------------------
   1 - access("PAY"."EMPID"="PER"."EMPID")
   2 - filter("PAY"."SALARY"<>199170)
```

Solution #8
Uncorrelated subquery (double negative)

```
SELECT per.empid, per.lname
FROM personnel per
WHERE per.empid NOT IN (SELECT pay.empid
FROM payroll pay
WHERE pay.salary != 199170);

Plan hash value: 2202369223
-------------------------------------------
| Id  | Operation                | Name      |
-------------------------------------------
|   0 | SELECT STATEMENT         |           |
|*  1 |  HASH JOIN RIGHT ANTI NA |           |
|*  2 |   TABLE ACCESS FULL      | PAYROLL   |
|   3 |   TABLE ACCESS FULL      | PERSONNEL |
-------------------------------------------
   1 - access("PER"."EMPID"="PAY"."EMPID")
   2 - filter("PAY"."SALARY"<>199170)
```

Comparison (11*g* Release 2)

METHOD	Plan Hash	Elapsed (us)	Buffer Gets
Uncorrelated subquery	3342999746	129	17
Correlated subquery	864898783	405	16
Relational algebra method	3901981856	426	16
Scalar subquery in the SELECT clause	750911849	701	16
Uncorrelated subquery (double negative)	2202369223	7702	241
Correlated subquery (double negative)	103534934	14499	241
Scalar subquery in the WHERE clause	3607962630	195999	10549
Aggregate function to check existence	3561519015	310690	10554

#8 STATISTICS ARE A DOUBLE-EDGED SWORD

"Oh, and by the way, could you please stop gathering statistics constantly? I don't know much about databases, but I do think I know the following: small tables tend to stay small, large tables tend to stay large, unique indexes have a tendency to stay unique, and non-unique indexes often stay non-unique."

—Dave Ensor

(as remembered by Mogens Norgaard)

Statistics: How and When?

November 2010 issue of the NoCOUG Journal

"Monitor the changes in execution plans and/or performance for the individual SQL statements … and perhaps as a consequence re-gather stats. That way, you'd leave stuff alone that works very well, thank you, and you'd put your efforts into exactly the things that have become worse."

—Mogens Norgaard
Statistics: How and When?
November 2010 issue of the NoCOUG Journal

“There are some statistics about your data that can be left unchanged for a long time, possibly forever; there are some statistics that need to be changed periodically; and there are some statistics that need to be changed constantly. … The biggest problem is that you need to understand the data.”

—Jonathan Lewis
Statistics: How and When?
November 2010 issue of the NoCOUG Journal

"It astonishes me how many shops prohibit any un-approved production changes and yet re-analyze schema stats weekly. Evidently, they do not understand that the [possible consequence] of schema re-analysis is to change their production SQL execution plans, and they act surprised when performance changes!"

—Don Burleson
Statistics: How and When?
November 2010 issue of the NoCOUG Journal

#9 PHYSICAL DATABASE DESIGN MATTERS

*“If a schema has no IOTs or clusters, that is a good indication that **no** thought has been given to the matter of optimizing data access.”*

—Tom Kyte (quoting Steve Adams)

Effective Oracle by Design (page 379)

The NoSQL complaint

“Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car.”

—incorrectly attributed to Esther Dyson

Editor of Release 1.0

“You can keep a car in a file cabinet because you can file the engine components in files in one drawer, and the axles and things in another, and keep a list of how everything fits together. You can, but you wouldn’t want to.”

—Esther Dyson

September 1989 issue of Release 1.0

Demonstration
(Refer to Physical Database Design.txt)

Based on the blog post The Twelve Days of NoSQL: Day Six: The False Premise of NoSQL

#10 SOMETIMES THE OPTIMIZER NEEDS A HINT

Who Said That?

"No optimizer is perfect and directives such as Oracle's hints provide the simplest workaround [in] situations in which the optimizer has chosen a suboptimal plan. Hints are useful tools not just to remedy an occasional suboptimal plan, but also for users who want to experiment with access paths, or simply have full control over the execution of a query."

1. Homer Simpson (star of The Simpsons TV show)
2. Oracle white paper
3. Tom Kyte (today's keynote speaker)
4. Dr. Edgar Codd (the inventor of relational database theory)

Suggestions from Dan Tow

- *Oracle's Cost-based Optimizer (CBO) does a perfectly good job on most SQL, requiring no manual tuning for most SQL.*
- *The CBO must parse quickly, use the data and indexes that it has, make assumptions about what it does not know, and deliver exactly the result that the SQL calls for.*
- *On a small fraction of the SQL, the constraints on the CBO result in a performance problem.*
- *Find SQL worth tuning, ignoring the great majority that already performs just fine.*
- *Find the true optimum execution plan (or at least one you verify is fast enough), manually, without the CBO's constraints or assumptions.*
- *Compare your manually chosen execution plan, and its resulting performance, with the CBO's plan and consider why the CBO did not select your plan, if it did not.*
- *Choose a solution that solves the problem.*

#11 AWR AND STATSPACK ARE A GOLDMINE OF HISTORICAL PERFORMANCE DATA

Demonstration

- 52 Weeks In the Life of a Database Graphs.pdf
- 52 Weeks in the Life of Another Database Graphs.pdf
- Contact Iggy for performance monitoring tool

#12 READERS DO NOT BLOCK WRITERS; WRITERS DO NOT BLOCK READERS

Serializability

"*To describe consistent transaction behavior when transactions run at the same time, database researchers have defined a transaction isolation model called serializability. The serializable mode of transaction behavior tries to ensure that transactions run in such a way that they* ***appear to be executed one at a time, or serially, rather than concurrently***."

—Oracle documentation up to 11*g* Release 1

Demonstration
(Refer to Serializability.txt)

Based on the blog post Day 12: The Twelve Days of SQL: Readers do not block writers; writers do not block readers

THROW AWAY THAT EXECUTION PLAN!

Demonstration

www.youtube.com/watch?v=ZVisY-fEoMw

#7 EXPLAIN PLAN LIES

Demonstration
(Refer to EXPLAIN PLAN lies.txt)

Based on the blog post Day 7: The Twelve Days of SQL: EXPLAIN PLAN lies

#5 THE QUERY COST IS ONLY AN ESTIMATE

#6 THE EXECUTION PLAN IS A TREE

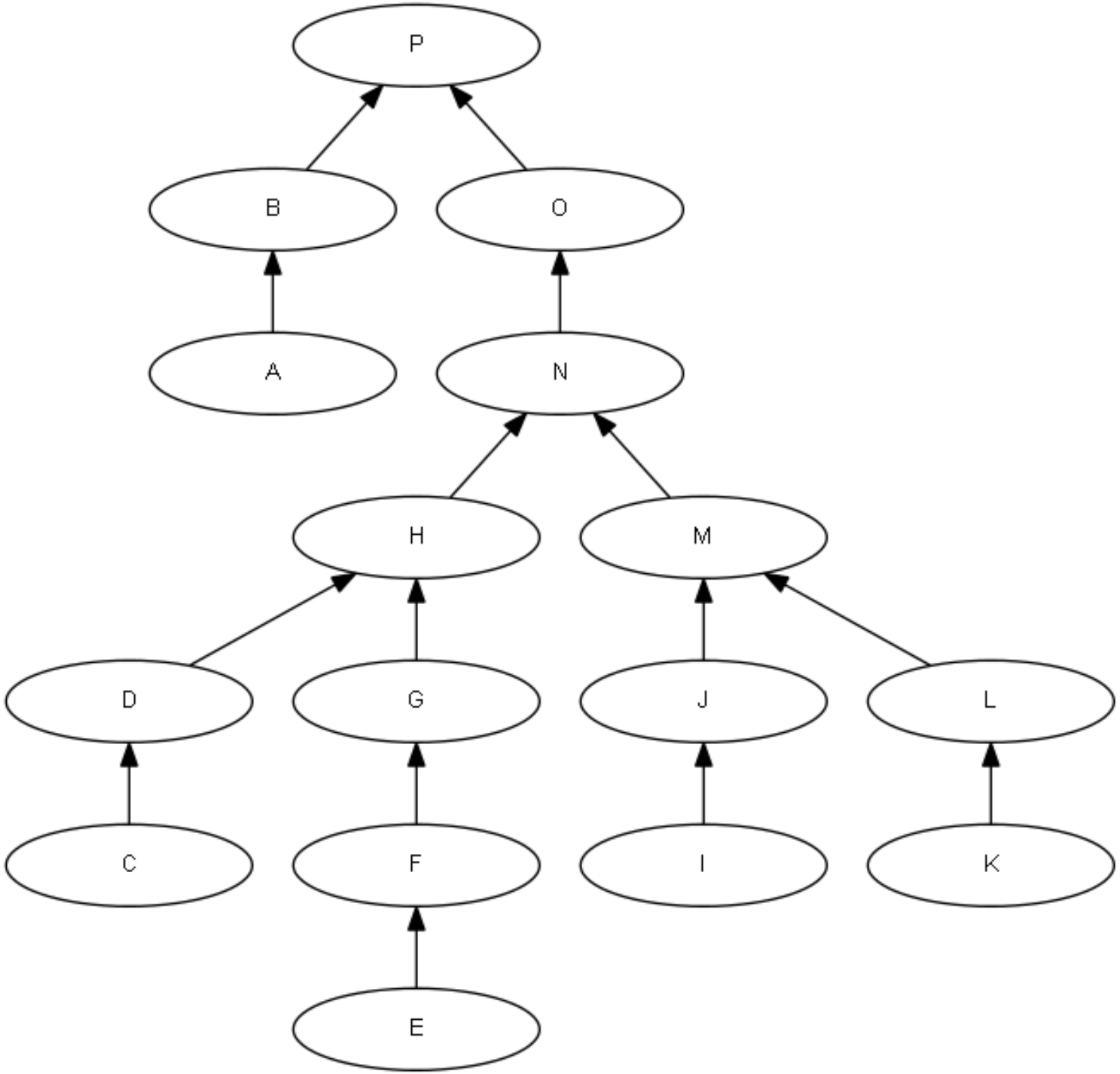
How to read EXPLAIN PLAN output (Not!)

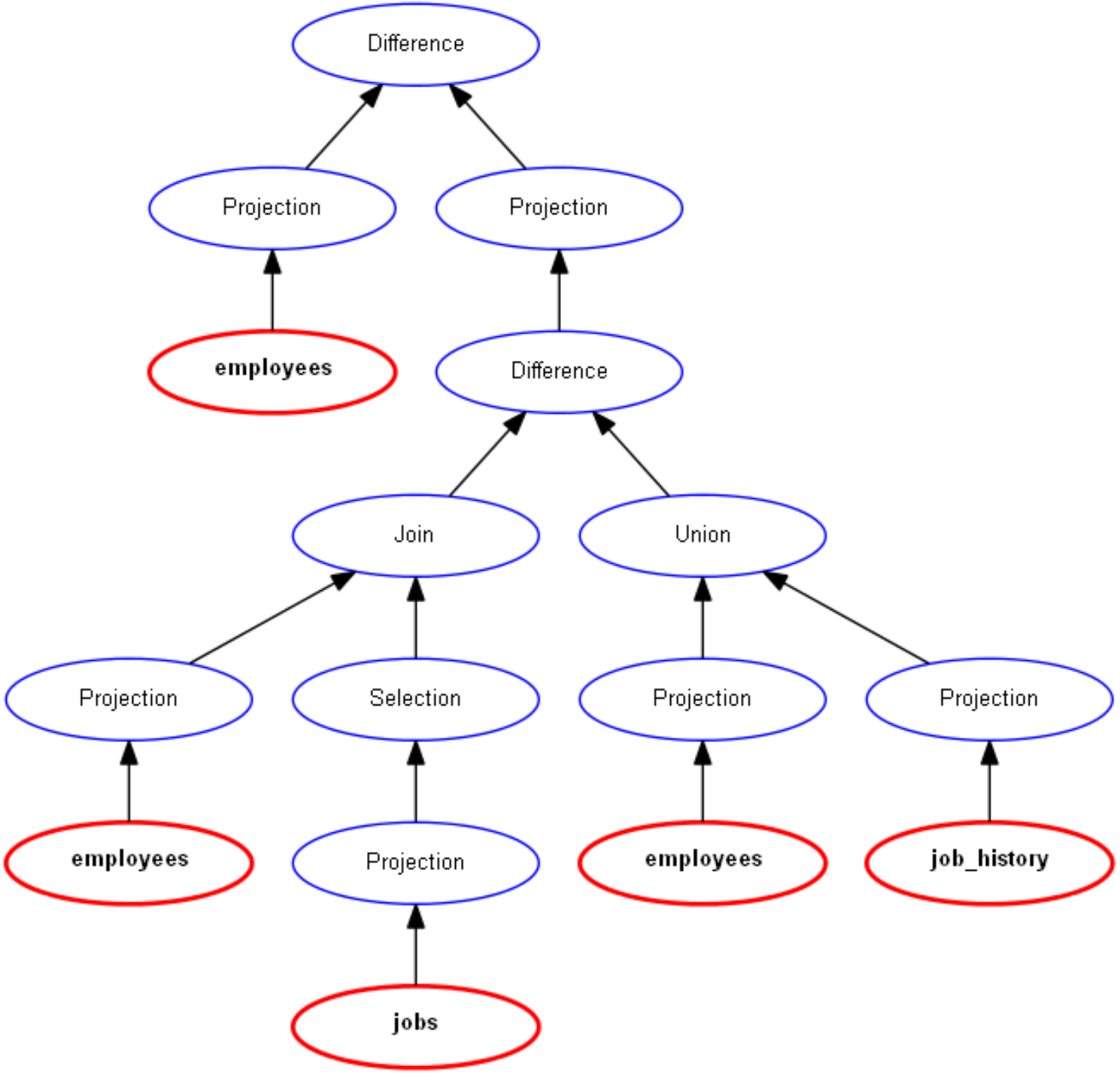
"The execution order in EXPLAIN PLAN output begins with the line that is the furthest indented to the right. The next step is the parent of that line. If two lines are indented equally, then the top line is normally executed first."

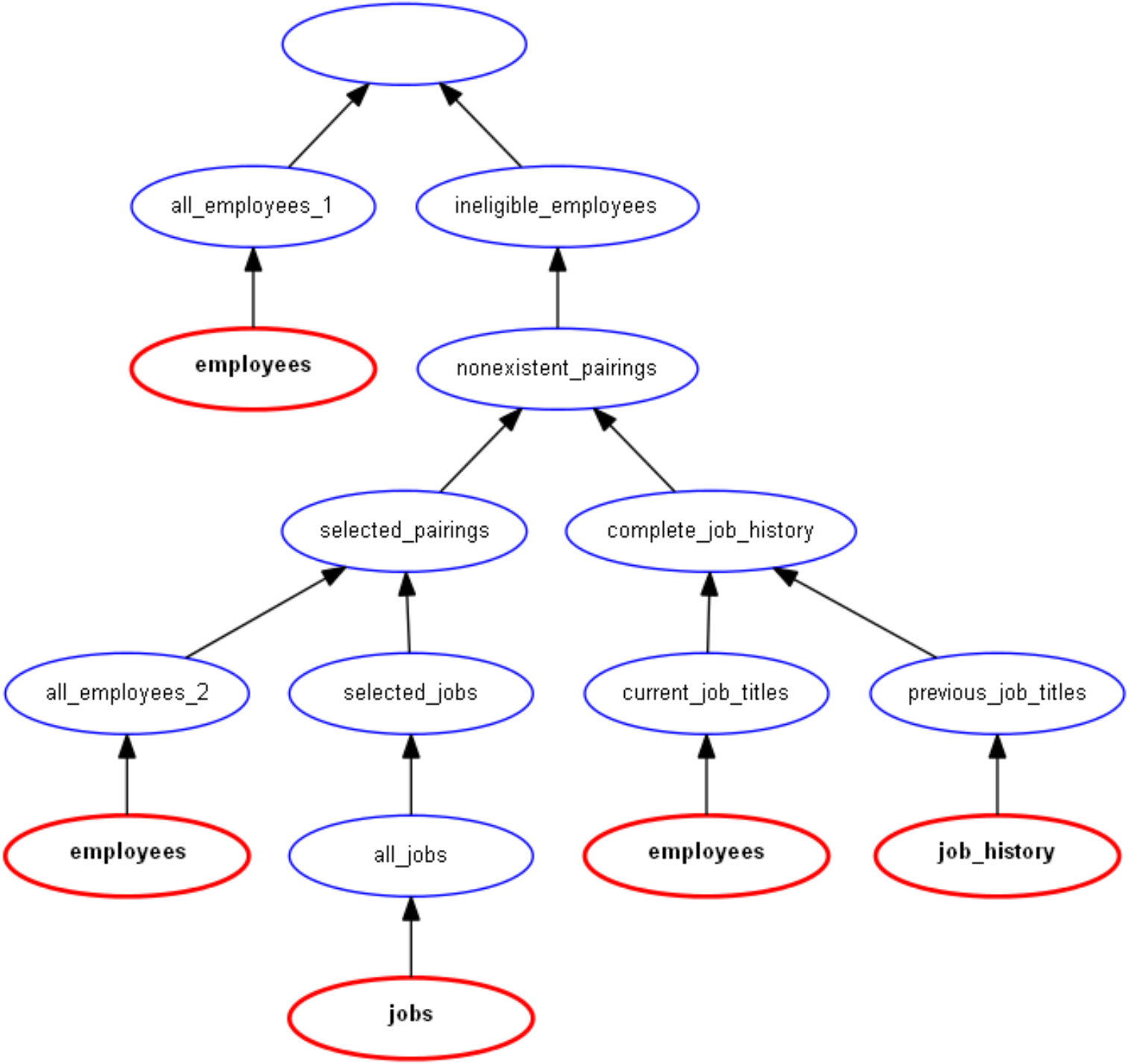
—Oracle documentation

The real scoop

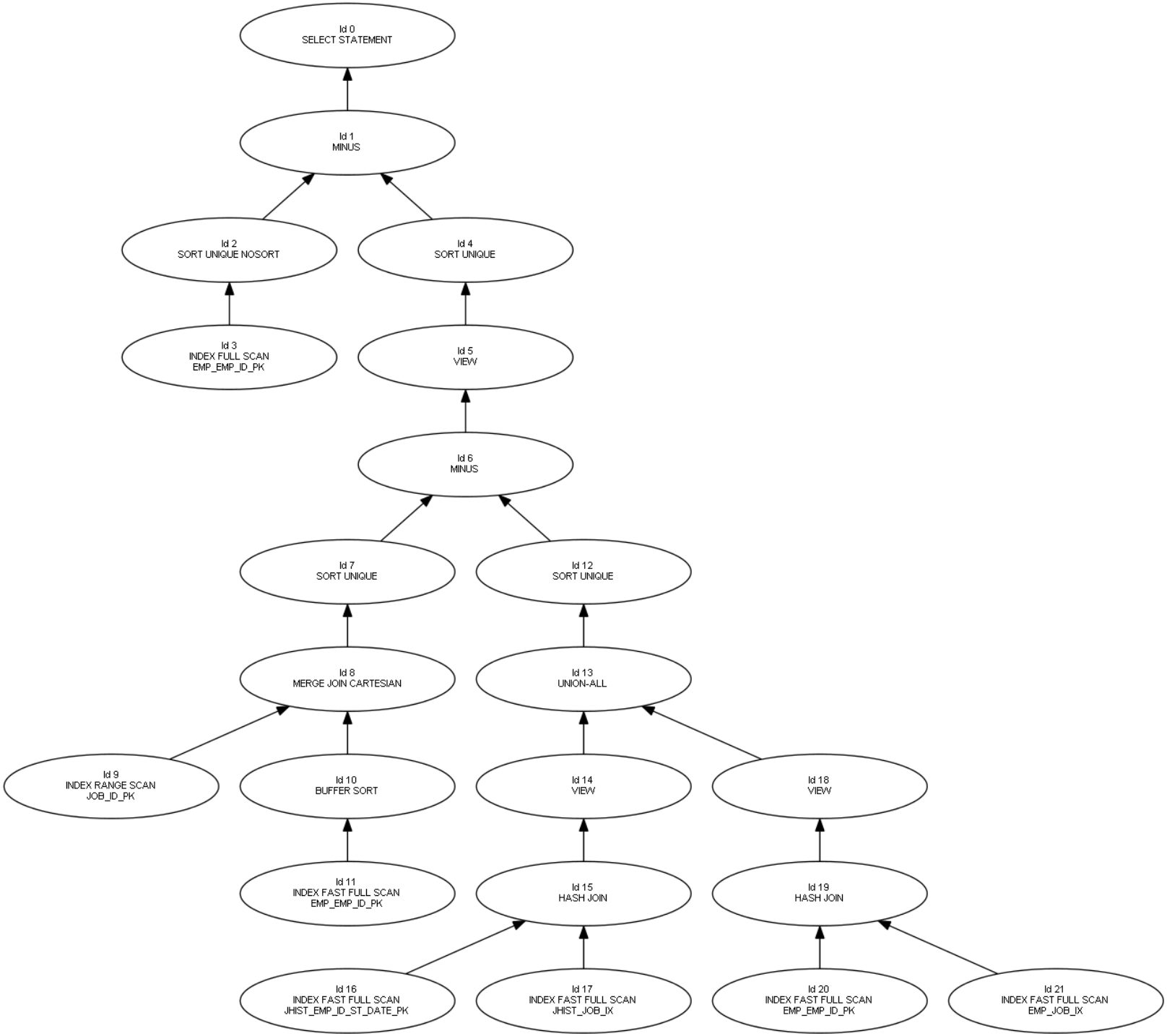
An Oracle EXPLAIN PLAN is a “tree” structure corresponding to a relational algebra expression. It is printed in “pre-order” sequence (visit the root of the tree, then traverse each subtree—if any—in pre-order sequence) but should be read in “post-order” sequence (first traverse each subtree—if any—in post-order sequence, then only visit the root of the tree).

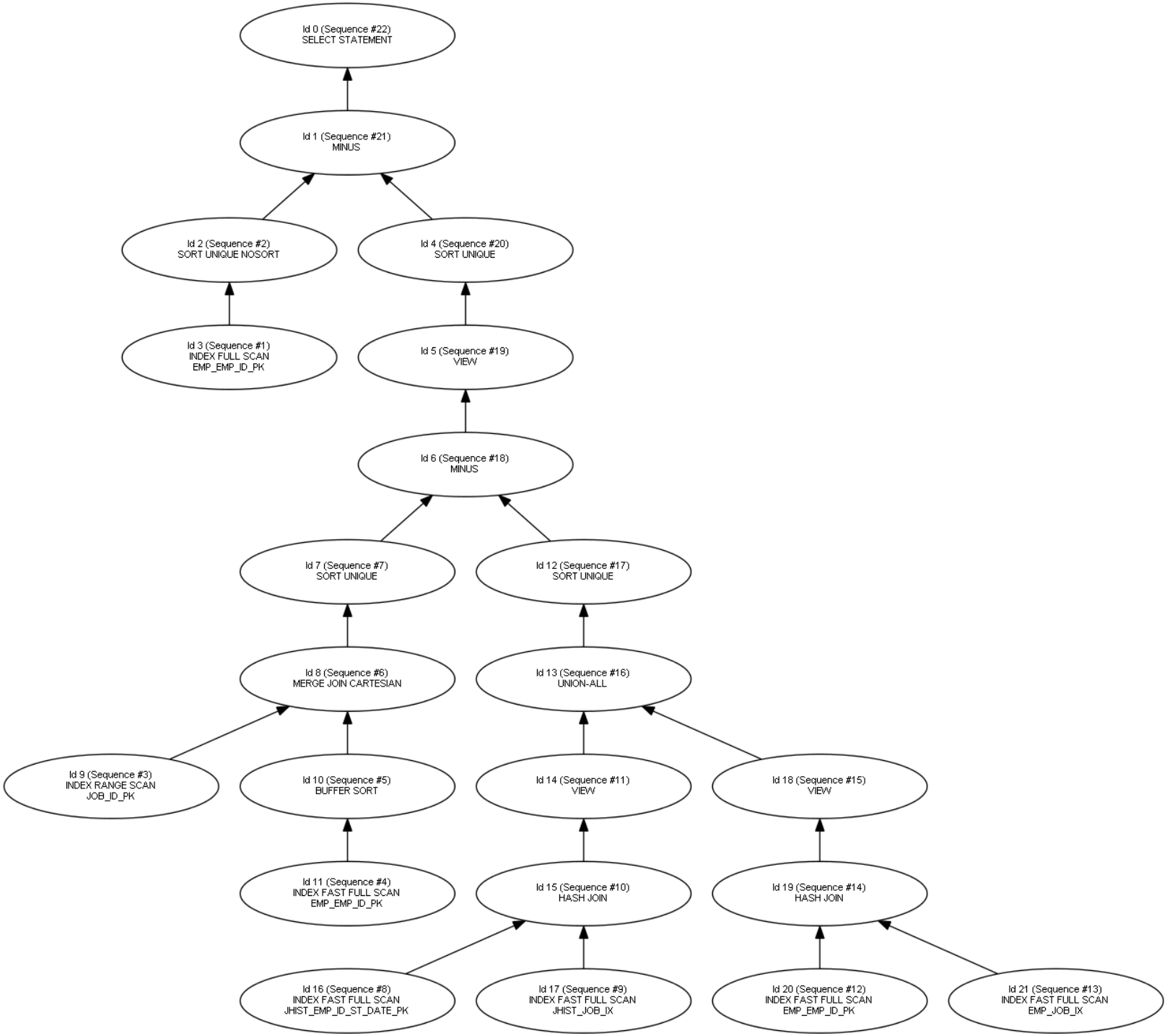

P
B
O
A
N
H
M
D
G
J
L
C
F
I
K
E


Difference
Projection
Projection
employees
Difference
Join
Union
Projection
Selection
Projection
Projection
employees
Projection
employees
job_history
jobs


all_employees_1
ineligible_employees
employees
nonexistent_pairings
selected_pairings
complete_job_history
all_employees_2
selected_jobs
current_job_titles
previous_job_titles
employees
all_jobs
employees
job_history
jobs

```
-------------------------------------------------------------------
| Id  | Operation                     | Name                      |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                           |
|   1 |  MINUS                        |                           |
|   2 |   SORT UNIQUE NOSORT          |                           |
|   3 |    INDEX FULL SCAN            | EMP_EMP_ID_PK             |
|   4 |   SORT UNIQUE                 |                           |
|   5 |    VIEW                       |                           |
|   6 |     MINUS                     |                           |
|   7 |      SORT UNIQUE              |                           |
|   8 |       MERGE JOIN CARTESIAN    |                           |
|   9 |        INDEX RANGE SCAN       | JOB_ID_PK                 |
|  10 |        BUFFER SORT            |                           |
|  11 |         INDEX FAST FULL SCAN  | EMP_EMP_ID_PK             |
|  12 |      SORT UNIQUE              |                           |
|  13 |       UNION-ALL               |                           |
|  14 |        VIEW                   | index$_join$_006          |
|  15 |         HASH JOIN             |                           |
|  16 |          INDEX FAST FULL SCAN| JHIST_EMP_ID_ST_DATE_PK   |
|  17 |          INDEX FAST FULL SCAN| JHIST_JOB_IX              |
|  18 |        VIEW                   | index$_join$_007          |
|  19 |         HASH JOIN             |                           |
|  20 |          INDEX FAST FULL SCAN| EMP_EMP_ID_PK             |
|  21 |          INDEX FAST FULL SCAN| EMP_JOB_IX                |
-------------------------------------------------------------------
```


Id 0
SELECT STATEMENT
Id 1
MINUS
Id 2
SORT UNIQUE NOSORT
Id 4
SORT UNIQUE
Id 3
INDEX FULL SCAN
EMP_EMP_ID_PK
Id 5
VIEW
Id 6
MINUS
Id 7
SORT UNIQUE
Id 12
SORT UNIQUE
Id 8
MERGE JOIN CARTESIAN
Id 13
UNION-ALL
Id 9
INDEX RANGE SCAN
JOB_ID_PK
Id 10
BUFFER SORT
Id 14
VIEW
Id 18
VIEW
Id 11
INDEX FAST FULL SCAN
EMP_EMP_ID_PK
Id 15
HASH JOIN
Id 19
HASH JOIN
Id 16
INDEX FAST FULL SCAN
JHIST_EMP_ID_ST_DATE_PK
Id 17
INDEX FAST FULL SCAN
JHIST_JOB_IX
Id 20
INDEX FAST FULL SCAN
EMP_EMP_ID_PK
Id 21
INDEX FAST FULL SCAN
EMP_JOB_IX


Id 0 (Sequence #22)
SELECT STATEMENT
Id 1 (Sequence #21)
MINUS
Id 2 (Sequence #2)
SORT UNIQUE NOSORT
Id 4 (Sequence #20)
SORT UNIQUE
Id 3 (Sequence #1)
INDEX FULL SCAN
EMP_EMP_ID_PK
Id 5 (Sequence #19)
VIEW
Id 6 (Sequence #18)
MINUS
Id 7 (Sequence #7)
SORT UNIQUE
Id 12 (Sequence #17)
SORT UNIQUE
Id 8 (Sequence #6)
MERGE JOIN CARTESIAN
Id 13 (Sequence #16)
UNION-ALL
Id 9 (Sequence #3)
INDEX RANGE SCAN
JOB_ID_PK
Id 10 (Sequence #5)
BUFFER SORT
Id 14 (Sequence #11)
VIEW
Id 18 (Sequence #15)
VIEW
Id 11 (Sequence #4)
INDEX FAST FULL SCAN
EMP_EMP_ID_PK
Id 15 (Sequence #10)
HASH JOIN
Id 19 (Sequence #14)
HASH JOIN
Id 16 (Sequence #8)
INDEX FAST FULL SCAN
JHIST_EMP_ID_ST_DATE_PK
Id 17 (Sequence #9)
INDEX FAST FULL SCAN
JHIST_JOB_IX
Id 20 (Sequence #12)
INDEX FAST FULL SCAN
EMP_EMP_ID_PK
Id 21 (Sequence #13)
INDEX FAST FULL SCAN
EMP_JOB_IX

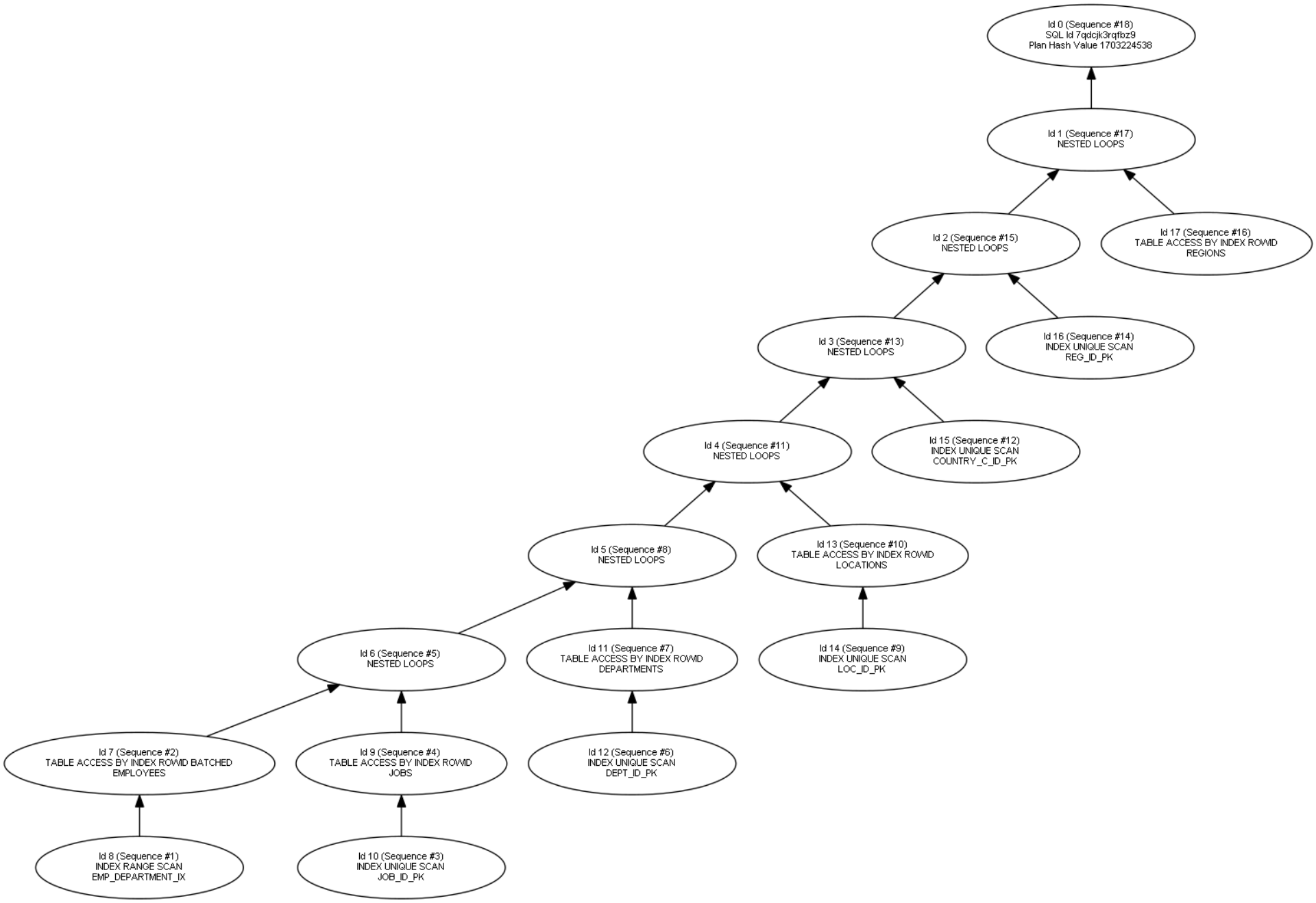
```
SELECT
  e.first_name, e.last_name, e.salary,
  j.job_title,
  d.department_name,
  l.city, l.state_province,
  c.country_name,
  r.region_name
FROM employees e, jobs j, departments d, locations l, countries c, regions r
WHERE
  e.department_id = 90
  AND j.job_id = e.job_id
  AND d.department_id = e.department_id
  AND l.location_id = d.location_id
  AND c.country_id = l.country_id
  AND r.region_id = c.region_id;
```

Hints for Deep Left Tree

```
LEADING (e j d l c r)
USE_NL(j)
USE_NL(d)
USE_NL(l)
USE_NL(c)
USE_NL(r)
```

Deep Left Tree

```
-----------------------------------------------------------
| Id  | Operation                          | Name           |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                   |                |
|   1 |  NESTED LOOPS                      |                |
|   2 |   NESTED LOOPS                     |                |
|   3 |    NESTED LOOPS                    |                |
|   4 |     NESTED LOOPS                   |                |
|   5 |      NESTED LOOPS                  |                |
|   6 |       NESTED LOOPS                 |                |
|   7 |        TABLE ACCESS BY INDEX ROWID| EMPLOYEES      |
|*  8 |         INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |
|   9 |        TABLE ACCESS BY INDEX ROWID| JOBS           |
|* 10 |         INDEX UNIQUE SCAN          | JOB_ID_PK      |
|  11 |       TABLE ACCESS BY INDEX ROWID | DEPARTMENTS    |
|* 12 |        INDEX UNIQUE SCAN           | DEPT_ID_PK     |
|  13 |      TABLE ACCESS BY INDEX ROWID   | LOCATIONS      |
|* 14 |       INDEX UNIQUE SCAN            | LOC_ID_PK      |
|* 15 |     INDEX UNIQUE SCAN              | COUNTRY_C_ID_PK   |
|* 16 |    INDEX UNIQUE SCAN               | REG_ID_PK      |
|  17 |   TABLE ACCESS BY INDEX ROWID      | REGIONS        |
-----------------------------------------------------------
```

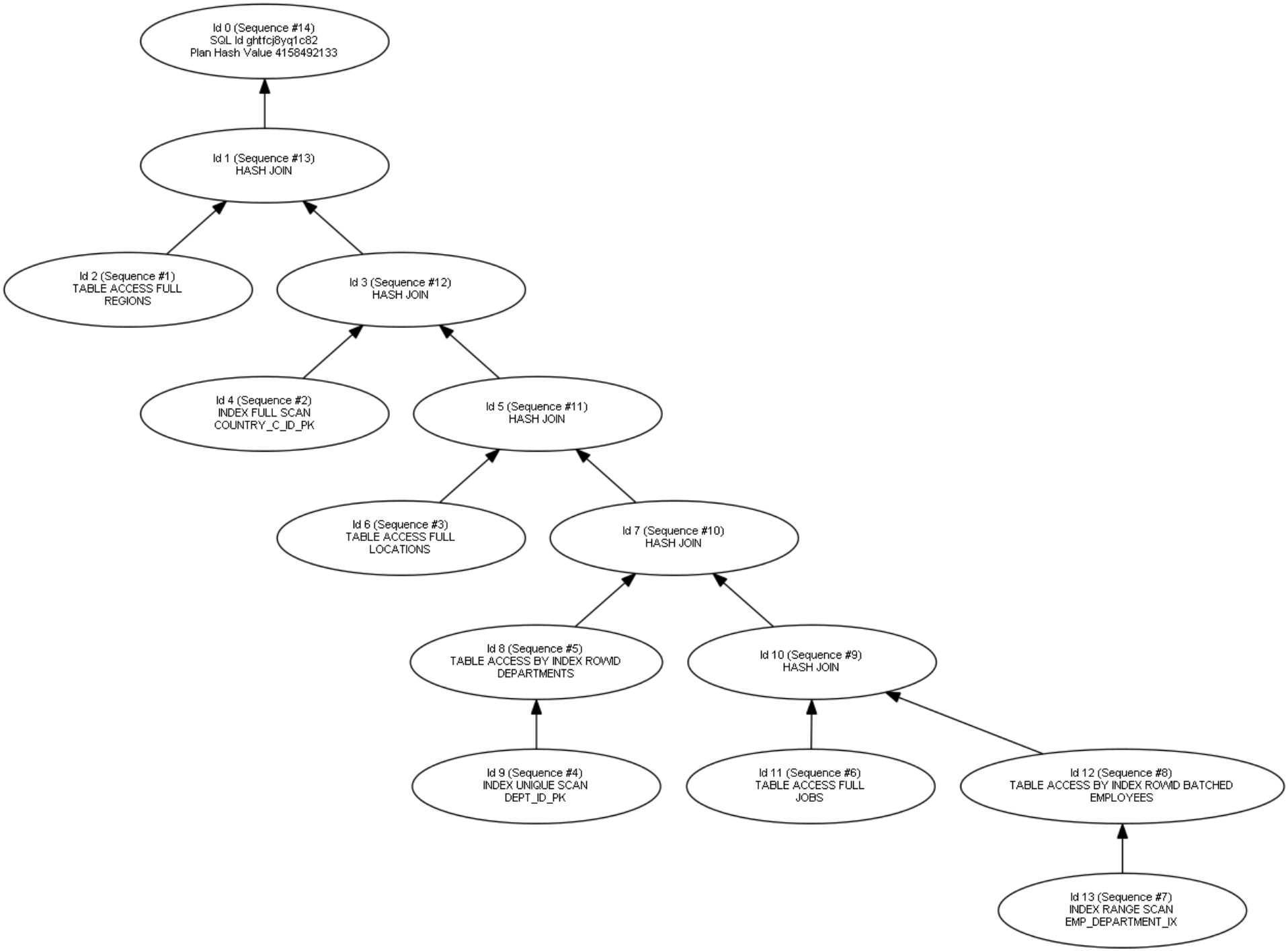

Id 0 (Sequence #18)
SQL Id 7qdcjk3rqfbz9
Plan Hash Value 1703224538
Id 1 (Sequence #17)
NESTED LOOPS
Id 2 (Sequence #15)
NESTED LOOPS
Id 17 (Sequence #16)
TABLE ACCESS BY INDEX ROWID
REGIONS
Id 3 (Sequence #13)
NESTED LOOPS
Id 16 (Sequence #14)
INDEX UNIQUE SCAN
REG_ID_PK
Id 4 (Sequence #11)
NESTED LOOPS
Id 15 (Sequence #12)
INDEX UNIQUE SCAN
COUNTRY_C_ID_PK
Id 5 (Sequence #8)
NESTED LOOPS
Id 13 (Sequence #10)
TABLE ACCESS BY INDEX ROWID
LOCATIONS
Id 6 (Sequence #5)
NESTED LOOPS
Id 11 (Sequence #7)
TABLE ACCESS BY INDEX ROWID
DEPARTMENTS
Id 14 (Sequence #9)
INDEX UNIQUE SCAN
LOC_ID_PK
Id 7 (Sequence #2)
TABLE ACCESS BY INDEX ROWID BATCHED
EMPLOYEES
Id 9 (Sequence #4)
TABLE ACCESS BY INDEX ROWID
JOBS
Id 12 (Sequence #6)
INDEX UNIQUE SCAN
DEPT_ID_PK
Id 8 (Sequence #1)
INDEX RANGE SCAN
EMP_DEPARTMENT_IX
Id 10 (Sequence #3)
INDEX UNIQUE SCAN
JOB_ID_PK

Hints for Deep Right Tree

```
LEADING (e j d l c r)
USE_HASH(j) SWAP_JOIN_INPUTS(j)
USE_HASH(d) SWAP_JOIN_INPUTS(d)
USE_HASH(l) SWAP_JOIN_INPUTS(l)
USE_HASH(c) SWAP_JOIN_INPUTS(c)
USE_HASH(r) SWAP_JOIN_INPUTS(r)
```

Deep Right Tree

```
-----------------------------------------------------------
| Id  | Operation                          | Name           |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                   |                |
|*  1 |  HASH JOIN                         |                |
|   2 |   TABLE ACCESS FULL                | REGIONS        |
|*  3 |   HASH JOIN                        |                |
|   4 |    INDEX FAST FULL SCAN            | COUNTRY_C_ID_PK|
|*  5 |    HASH JOIN                       |                |
|   6 |     TABLE ACCESS FULL              | LOCATIONS      |
|*  7 |     HASH JOIN                      |                |
|   8 |      TABLE ACCESS BY INDEX ROWID   | DEPARTMENTS    |
|*  9 |       INDEX UNIQUE SCAN            | DEPT_ID_PK     |
|* 10 |      HASH JOIN                     |                |
|  11 |       TABLE ACCESS FULL            | JOBS           |
|  12 |       TABLE ACCESS BY INDEX ROWID  | EMPLOYEES      |
|* 13 |        INDEX RANGE SCAN            | EMP_DEPARTMENT_IX|
-----------------------------------------------------------
```

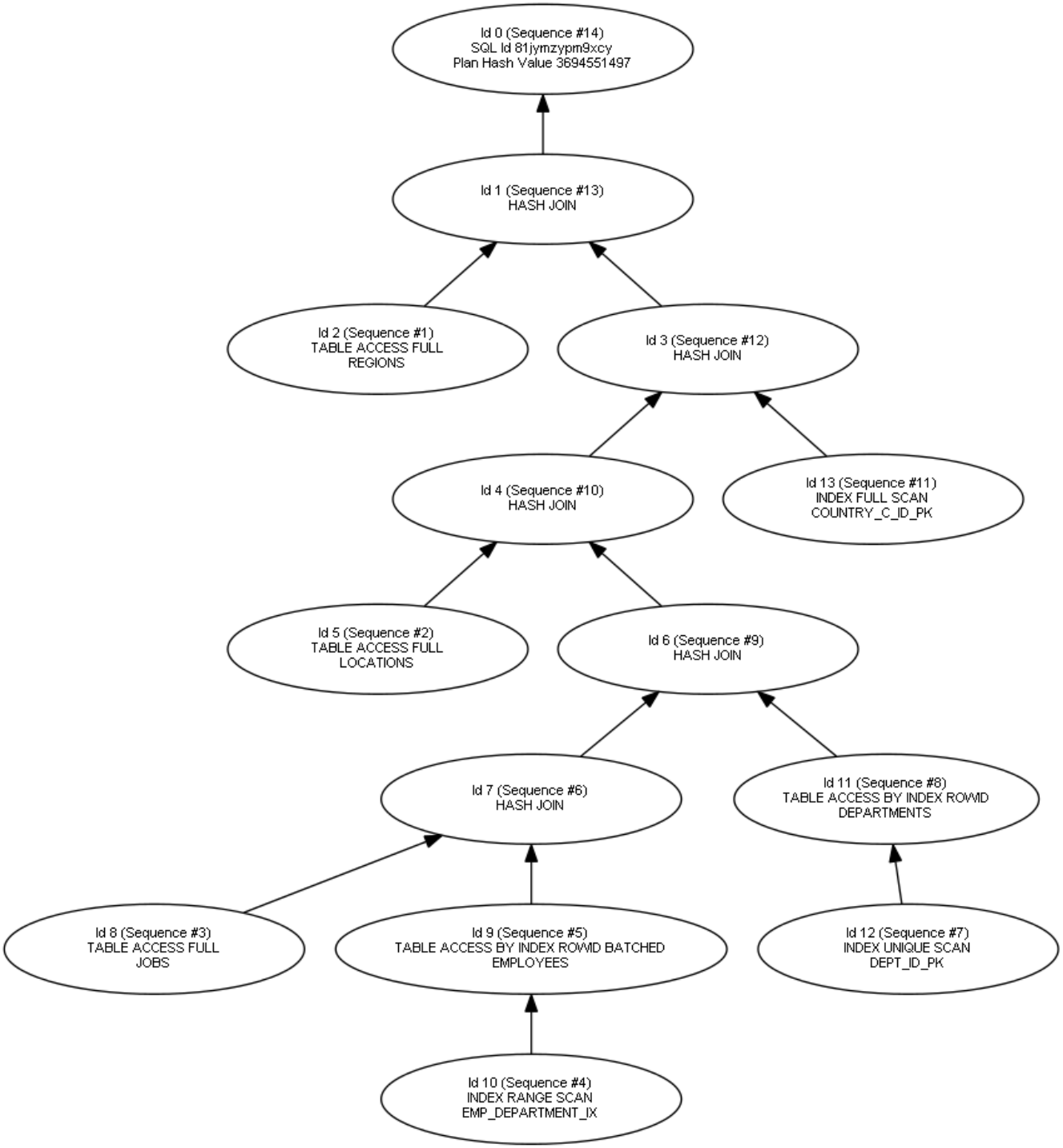

Id 0 (Sequence #14)
SQL Id ghtfcj8yq1c82
Plan Hash Value 4158492133
Id 1 (Sequence #13)
HASH JOIN
Id 2 (Sequence #1)
TABLE ACCESS FULL
REGIONS
Id 3 (Sequence #12)
HASH JOIN
Id 4 (Sequence #2)
INDEX FULL SCAN
COUNTRY_C_ID_PK
Id 5 (Sequence #11)
HASH JOIN
Id 6 (Sequence #3)
TABLE ACCESS FULL
LOCATIONS
Id 7 (Sequence #10)
HASH JOIN
Id 8 (Sequence #5)
TABLE ACCESS BY INDEX ROWID
DEPARTMENTS
Id 10 (Sequence #9)
HASH JOIN
Id 9 (Sequence #4)
INDEX UNIQUE SCAN
DEPT_ID_PK
Id 11 (Sequence #6)
TABLE ACCESS FULL
JOBS
Id 12 (Sequence #8)
TABLE ACCESS BY INDEX ROWID BATCHED
EMPLOYEES
Id 13 (Sequence #7)
INDEX RANGE SCAN
EMP_DEPARTMENT_IX

Hints for Zig-Zag Tree

```
LEADING (e j d l c r)
USE_HASH(j) SWAP_JOIN_INPUTS(j)
USE_HASH(d) NO_SWAP_JOIN_INPUTS(d)
USE_HASH(l) SWAP_JOIN_INPUTS(l)
USE_HASH(c) NO_SWAP_JOIN_INPUTS(c)
USE_HASH(r) SWAP_JOIN_INPUTS(r)
```

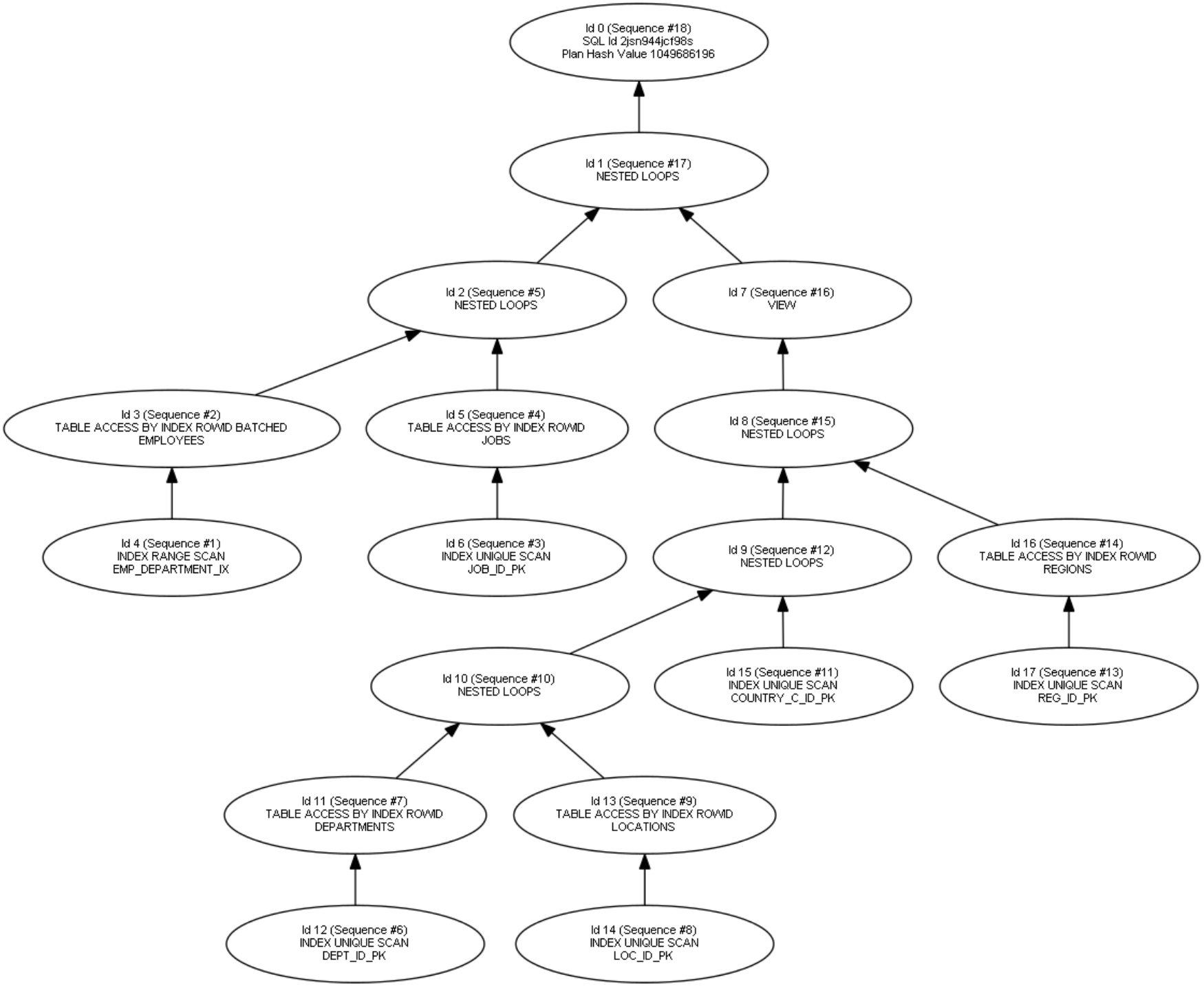
EXPLAIN PLAN for Zig-Zag Tree

```
-----------------------------------------------------------
| Id  | Operation                      | Name               |
-----------------------------------------------------------
|   0 | SELECT STATEMENT               |                    |
|*  1 |  HASH JOIN                     |                    |
|   2 |   TABLE ACCESS FULL            | REGIONS            |
|*  3 |   HASH JOIN                    |                    |
|*  4 |    HASH JOIN                   |                    |
|   5 |     TABLE ACCESS FULL          | LOCATIONS          |
|*  6 |     HASH JOIN                  |                    |
|*  7 |      HASH JOIN                 |                    |
|   8 |       TABLE ACCESS FULL        | JOBS               |
|   9 |       TABLE ACCESS BY INDEX ROWID| EMPLOYEES          |
|* 10 |        INDEX RANGE SCAN        | EMP_DEPARTMENT_IX  |
|  11 |      TABLE ACCESS BY INDEX ROWID | DEPARTMENTS        |
|* 12 |       INDEX UNIQUE SCAN        | DEPT_ID_PK         |
|  13 |    INDEX FAST FULL SCAN        | COUNTRY_C_ID_PK    |
-----------------------------------------------------------
```

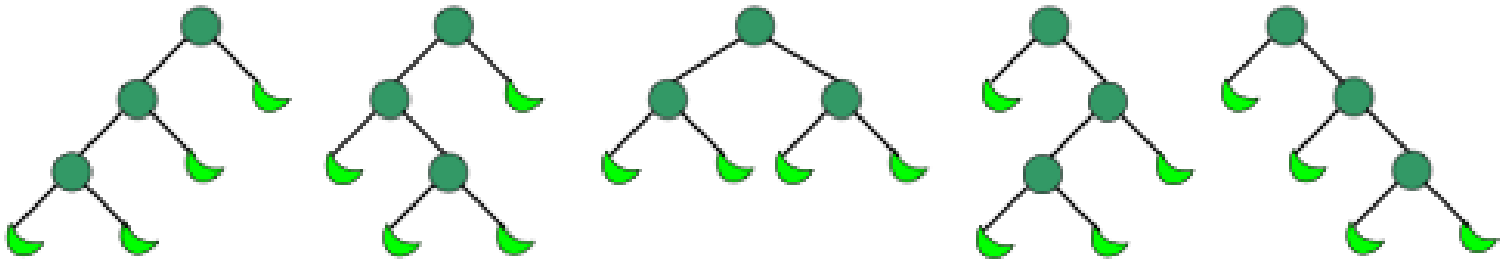

Id 0 (Sequence #14)
SQL Id 81jymzypm9xcy
Plan Hash Value 3694551497
Id 1 (Sequence #13)
HASH JOIN
Id 2 (Sequence #1)
TABLE ACCESS FULL
REGIONS
Id 3 (Sequence #12)
HASH JOIN
Id 4 (Sequence #10)
HASH JOIN
Id 13 (Sequence #11)
INDEX FULL SCAN
COUNTRY_C_ID_PK
Id 5 (Sequence #2)
TABLE ACCESS FULL
LOCATIONS
Id 6 (Sequence #9)
HASH JOIN
Id 7 (Sequence #6)
HASH JOIN
Id 11 (Sequence #8)
TABLE ACCESS BY INDEX ROWID
DEPARTMENTS
Id 8 (Sequence #3)
TABLE ACCESS FULL
JOBS
Id 9 (Sequence #5)
TABLE ACCESS BY INDEX ROWID BATCHED
EMPLOYEES
Id 12 (Sequence #7)
INDEX UNIQUE SCAN
DEPT_ID_PK
Id 10 (Sequence #4)
INDEX RANGE SCAN
EMP_DEPARTMENT_IX

Bushy Tree (Inline view with NO_MERGE)

```
SELECT /*+ LEADING(e j d) USE_NL(j) USE_NL(d) */
  e.first_name, e.last_name, e.salary,
  j.job_title,
  d.department_name, d.city, d.state_province, d.country_name, d.region_name
FROM
  employees e,
  jobs j,
  (
    SELECT /*+ NO_MERGE */
      d.department_id, d.department_name,
      l.city, l.state_province,
      c.country_name,
      r.region_name
    FROM departments d, locations l, countries c, regions r
    WHERE l.location_id = d.location_id
    AND c.country_id = l.country_id
    AND r.region_id = c.region_id
  ) d
WHERE e.department_id = 90
AND j.job_id = e.job_id
AND d.department_id = e.department_id;
```


Id 0 (Sequence #18)
SQL Id 2jsn944jcf98s
Plan Hash Value 1049686196
Id 1 (Sequence #17)
NESTED LOOPS
Id 2 (Sequence #5)
NESTED LOOPS
Id 7 (Sequence #16)
VIEW
Id 3 (Sequence #2)
TABLE ACCESS BY INDEX ROWID BATCHED
EMPLOYEES
Id 5 (Sequence #4)
TABLE ACCESS BY INDEX ROWID
JOBS
Id 8 (Sequence #15)
NESTED LOOPS
Id 4 (Sequence #1)
INDEX RANGE SCAN
EMP_DEPARTMENT_IX
Id 6 (Sequence #3)
INDEX UNIQUE SCAN
JOB_ID_PK
Id 9 (Sequence #12)
NESTED LOOPS
Id 16 (Sequence #14)
TABLE ACCESS BY INDEX ROWID
REGIONS
Id 10 (Sequence #10)
NESTED LOOPS
Id 15 (Sequence #11)
INDEX UNIQUE SCAN
COUNTRY_C_ID_PK
Id 17 (Sequence #13)
INDEX UNIQUE SCAN
REG_ID_PK
Id 11 (Sequence #7)
TABLE ACCESS BY INDEX ROWID
DEPARTMENTS
Id 13 (Sequence #9)
TABLE ACCESS BY INDEX ROWID
LOCATIONS
Id 12 (Sequence #6)
INDEX UNIQUE SCAN
DEPT_ID_PK
Id 14 (Sequence #8)
INDEX UNIQUE SCAN
LOC_ID_PK

Bushy Trees



Total Number of Trees

N	FACTORIAL(N)	CATALAN(N-1)	Total trees
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280
8	40,320	429	17,297,280
9	362,880	1,430	518,918,400
10	3,628,800	4,862	17,643,225,600

Thank you for listening!

iggy_fernandez@hotmail.com

The Hitchhiker's Guide to the EXPLAIN PLAN

NoCOUG Journal Archive