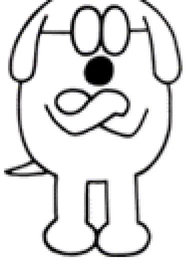
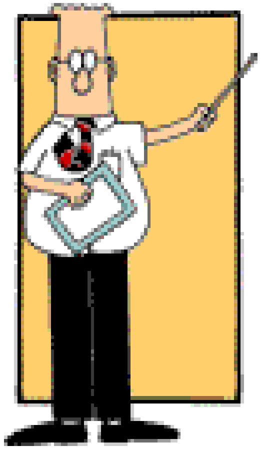
Execution

Oracle SQL ~~Tuning~~
An Introduction



Overview

- Foundation
  - Optimizer, cost vs. rule, data storage, SQL-execution phases, ...
- Creating & reading execution plans
  - Access paths, single table, joins, ...
- Utilities
  - Tracefiles, SQL hints, analyze/dbms_stat
- Warehouse specifics
  - Star queries & bitmap indexing
  - ETL
- Availability in version 9, 10, 11 ?



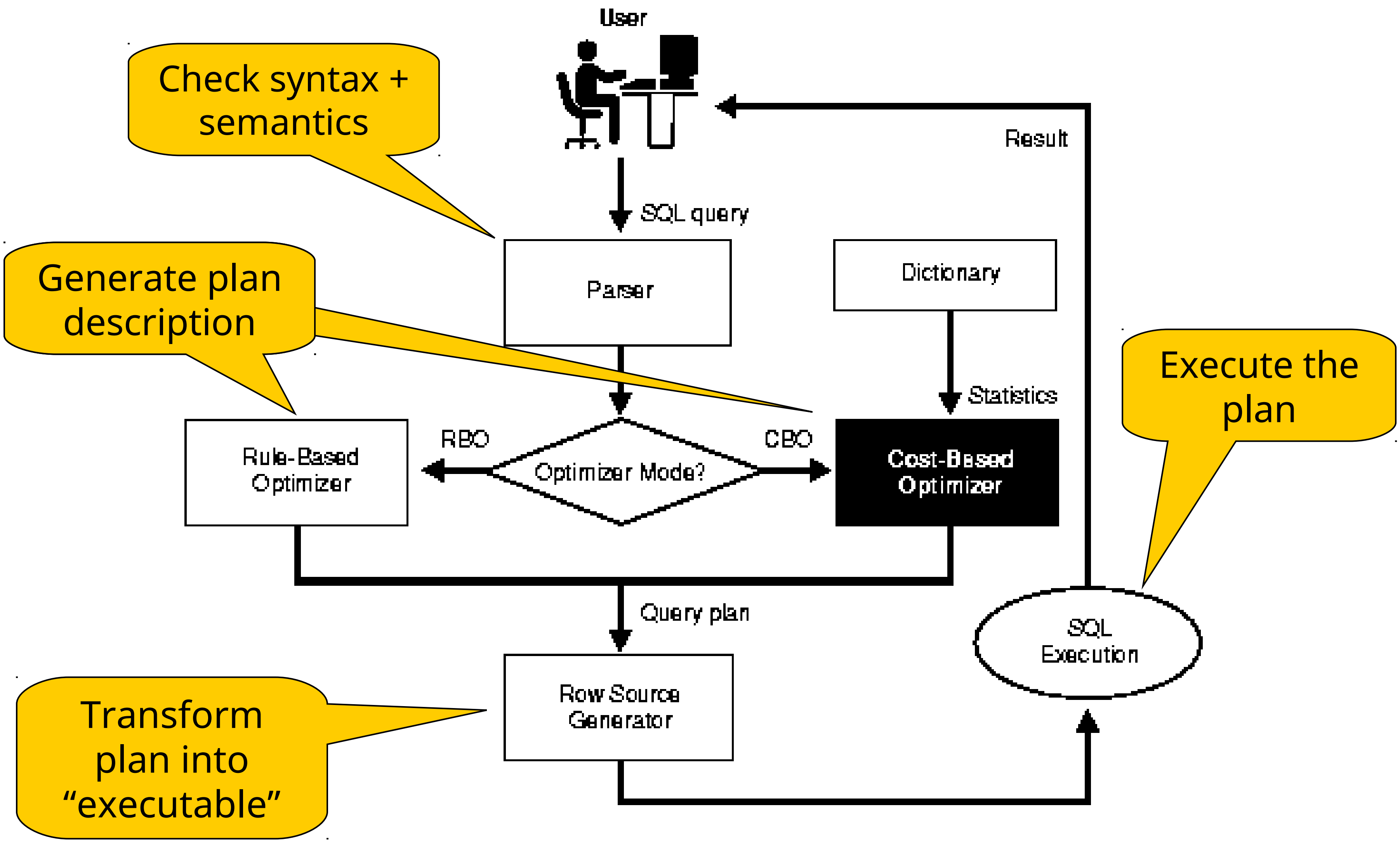
Goals

- Read execution plans
  - Table access
  - Index access
  - Joins
  - Subqueries
- Understand execution plans
  - Understand performance
  - Basic understanding of SQL optimization
- Start thinking how <u>you</u> should have executed it

Next…

- Basic Concepts (13)
  - Background information
- SQL-Execution (50)
  - Read + understand

Optimizer Overview



Cost vs. Rule

- Rule (RBO: Rule Based Optimization)
  - Hardcoded heuristic rules determine plan
    - “Access via index is better than full table scan”
    - “Fully matched index is better than partially matched index”
    - …
- Cost (2 modes)
  - Statistics of data play role in plan determination
    - Best throughput mode: retrieve **all rows** asap
      - First compute, then return fast
    - Best response mode: retrieve **first row** asap
      - Start returning while computing (if possible)

How to set which one?

- Instance level: Optimizer_Mode parameter
  - Rule
  - Choose
    - if statistics then CBO (all_rows), else RBO
  - First_rows, First_rows_n (1, 10, 100, 1000)
  - All_rows
- Session level:
  - Alter session set optimizer_mode=<mode>;
- Statement level:
  - Hints inside SQL text specify mode to be used

DML vs. Queries

- Open => Parse => Execute (=> Fetch^n)

```
SELECT ename,salary
FROM emp
WHERE salary>100000
```

Fetches done
By client

Same SQL optimization

```
UPDATE emp
SET commission='N'
WHERE salary>100000
```

All fetches done internally
by SQL-Executor

CLIENT => SQL => <= Data or Returncode<= **SERVER**

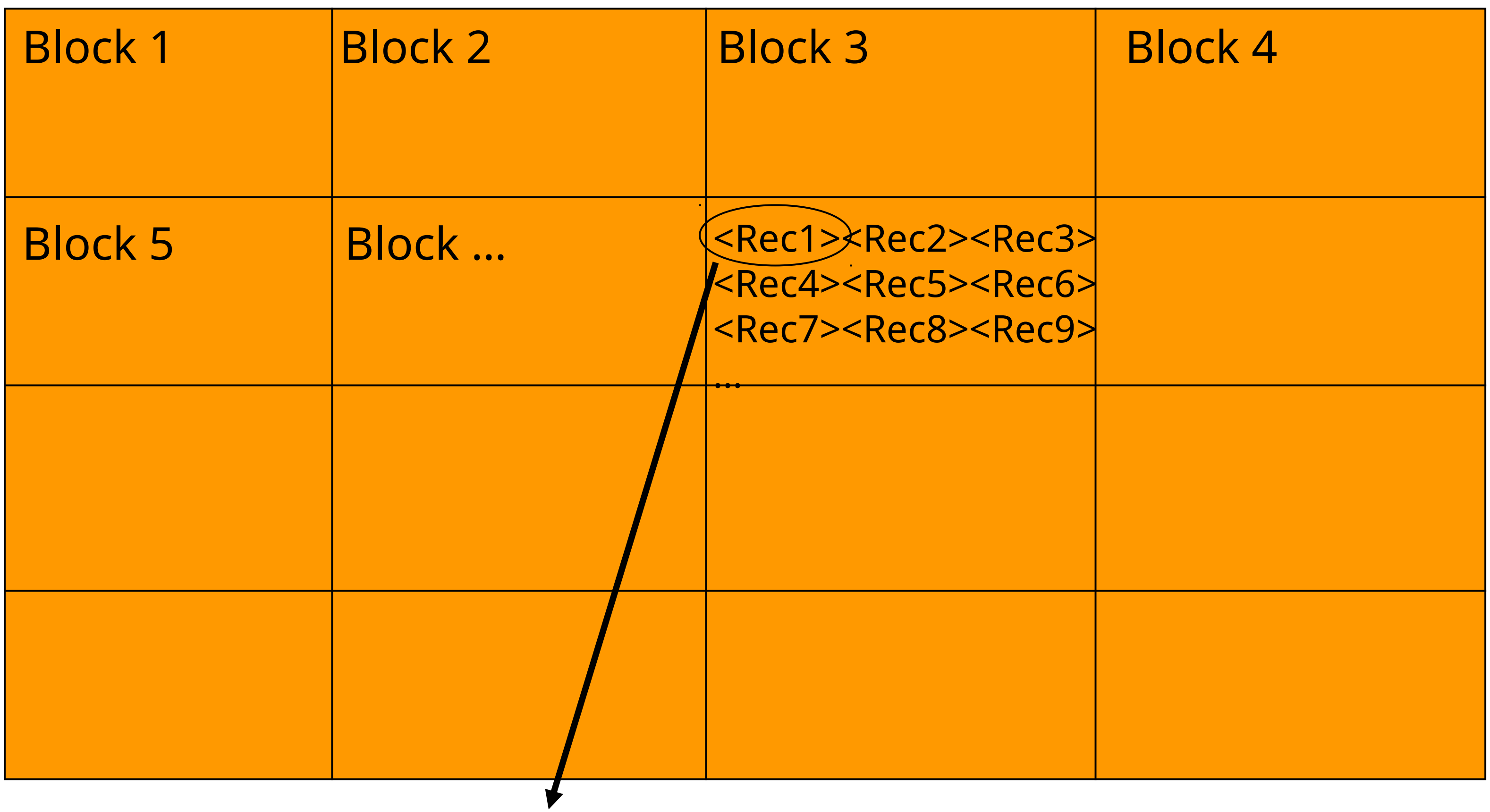
Data Storage: Tables

- Oracle stores all data inside datafiles
  - Location & size determined by DBA
  - Logically grouped in tablespaces
  - Each file is identified by a relative file number (fno)
- Datafile consists of data-blocks
  - Size equals value of *db_block_size* parameter
  - Each block is identified by its offset in the file
- Data-blocks contain rows
  - Each row is identified by its sequence in the block

ROWID: <Block>.<Row>.<File>

Data Storage: Tables

File x

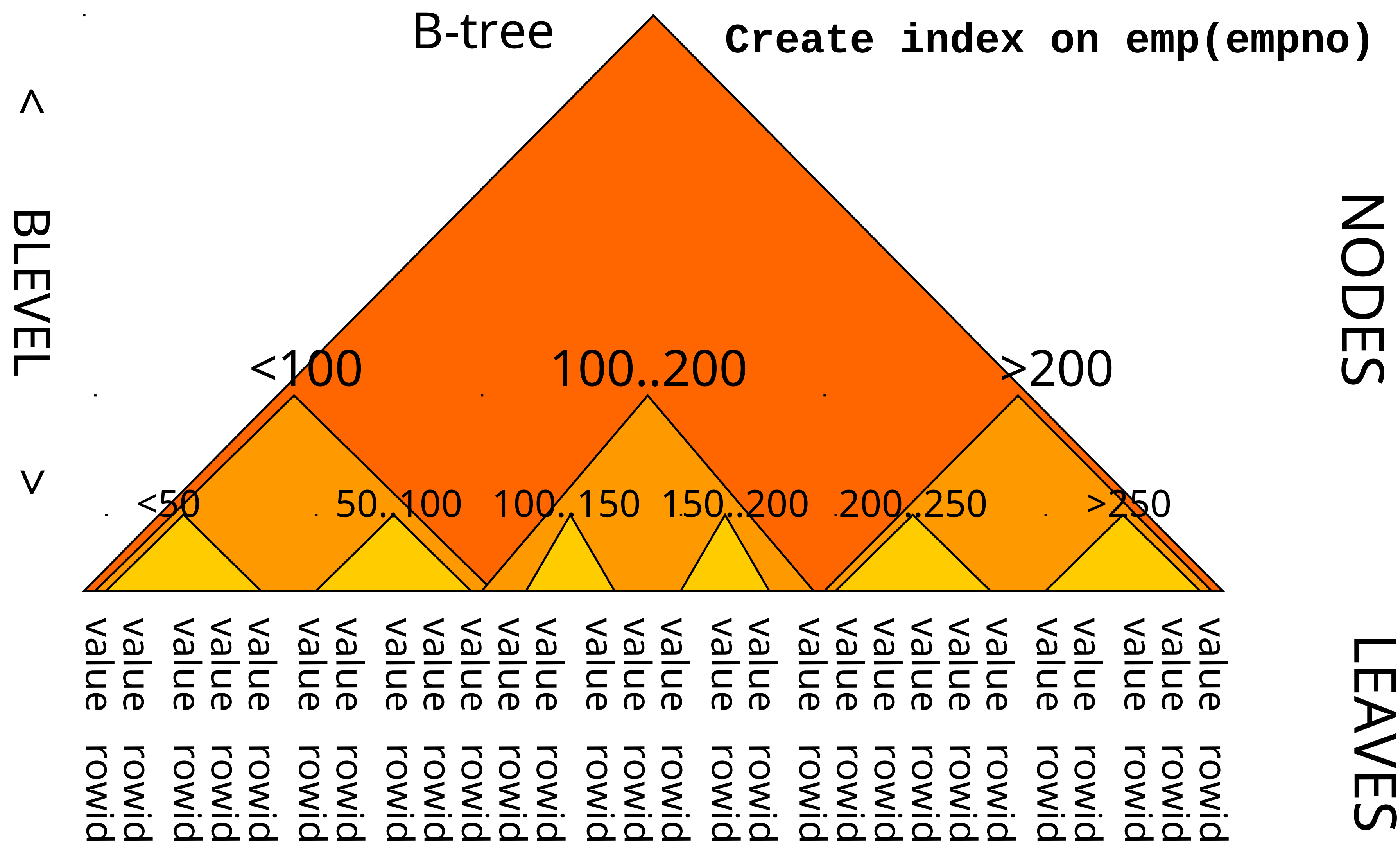


Rowid: 00000006.0000.000X

Data Storage: Indexes

- Balanced trees
  - Indexed column(s) sorted and stored seperately
    - NULL values are excluded (not added to the index)
  - Pointer structure enables logarithmic search
    - Access index first, find pointer to table, then access table
- B-trees consist of
  - Node blocks
    - Contain pointers to other node, or leaf blocks
  - Leaf blocks
    - Contain actual indexed values
    - Contain rowids (pointer to rows)
- Also stored in blocks in datafiles
  - Proprietary format

Data Storage: Indexes



Data Storage: Indexes

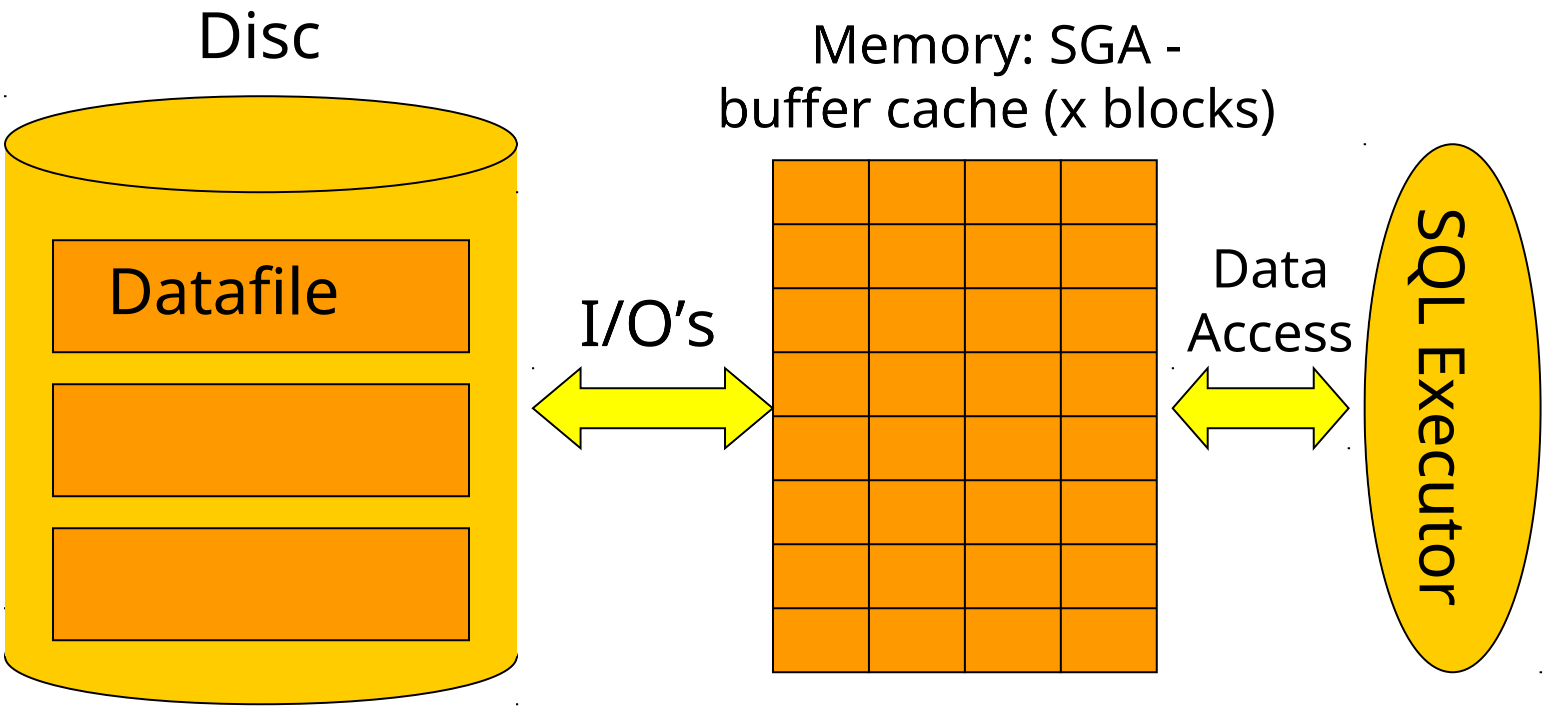
Datafile

Block 1	Block 2	Block 3	Block 4
Block 5	Block …	Index Node Block	Index Leaf Block
Index Leaf Block			

No particular order of node and leaf blocks

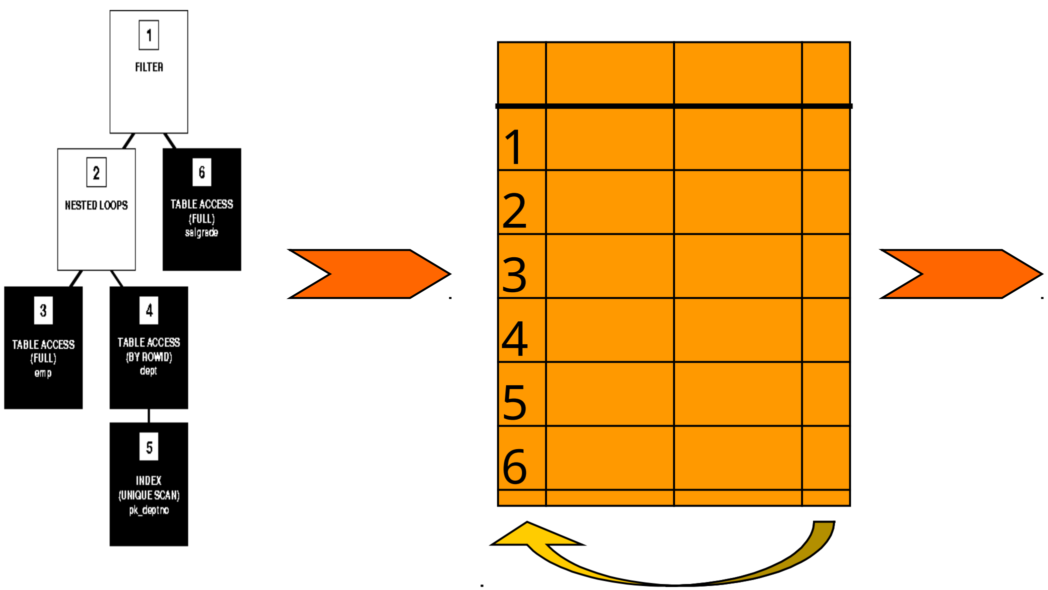
Table & Index I/O

- I/O's are done at 'block level'
  - LRU list controls who 'makes place' in the cache



Explain Plan Utility

- "Explain plan for <SQL-statement>"
  - Stores plan (row-sources + operations) in Plan_Table
  - View on Plan_Table (or 3rd party tool) formats into readable plan



>Filter
>....NL
>........TA-full
>........TA-rowid
>...........Index Uscan
>....TA-full

Explain Plan Utility

```
create table PLAN_TABLE (
  statement_id     varchar2(30),   operation        varchar2(30),
  options          varchar2(30),   object_owner     varchar2(30),
  object_name      varchar2(30),   id               numeric,
  parent_id        numeric,        position         numeric,
  cost             numeric,        bytes            numeric);
```

```
create or replace view PLANS(STATEMENT_ID,PLAN,POSITION) as
select statement_id,
  rpad('>',2*level,'.')||operation||
  decode(options,NULL,'',' (')||nvl(options,' ')||
  decode(options,NULL,'',') ')||
  decode(object_owner,NULL,'',object_owner||'.')||object_name plan,
  position
from plan_table
start with id=0
connect by prior id=parent_id
       and prior nvl(statement_id,'NULL')=nvl(statement_id,'NULL')
```

Execution Plans

1. Single table without index
2. Single table with index
3. Joins
   1. Nested Loop
   2. Sort Merge
   3. Hash1 (small/large), hash2 (large/large)
4. Special operators

Single Table, no Index (1.1)

```
SELECT *
FROM emp;
```

```
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

- Full table scan (FTS)
  - All blocks read sequentially into buffer cache
    - Also called “buffer-gets”
    - Done via multi-block I/O’s (db_file_multiblock_read_count)
    - Till high-water-mark reached (truncate resets, delete not)
  - Per block: extract + return all rows
    - Then put block at LRU-end of LRU list (!)
    - All other operations put block at MRU-end

Single Table, no Index (1.2)

```
SELECT *
FROM emp
WHERE sal > 100000;
```

```
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

- Full table scan with filtering
  - Read all blocks
  - Per block extract, filter, then return row
    - **Simple where-clause filters never shown in plan**
    - FTS with: rows-in > rows-out

Single Table, no Index (1.3)

```
SELECT *
FROM emp
ORDER BY ename;
```

```
>.SELECT STATEMENT
>...SORT order by
>.....TABLE ACCESS full emp
```

- FTS followed by sort on ordered-by column(s)
  - "Followed by" Ie. SORT won't return rows to its parent row-source till its child row-source fully completed
  - SORT order by: rows-in = rows-out
  - Small sorts done in memory (SORT_AREA_SIZE)
  - Large sorts done via TEMPORARY tablespace
    - Potentially many I/O's

Single Table, no Index (1.3)

```
SELECT *
FROM emp
ORDER BY ename;

Emp(ename)
```

```
>.SELECT STATEMENT
>...TABLE ACCESS full emp
>.....INDEX full scan i_emp_ename
```

- If ordered-by column(s) is indexed
  - Index Full Scan
  - CBO uses index if mode = First_Rows
  - If index is used => no sort is necessary

Single Table, no Index (1.4)

```
SELECT job,sum(sal)
FROM emp
GROUP BY job;
```

```
>.SELECT STATEMENT
>...SORT group by
>.....TABLE ACCESS full emp
```

- FTS followed by sort on grouped-by column(s)
  - FTS will only retrieve job and sal columns
    - Small intermediate rowlength => sort more likely in memory
  - SORT group by: rows-in >> rows-out
  - Sort also computes aggregates

Single Table, no Index (1.5)

```
SELECT job,sum(sal)
FROM emp
GROUP BY job
HAVING sum(sal)>200000;
```

```
>.SELECT STATEMENT
>...FILTER
>.....SORT group by
>.......TABLE ACCESS full emp
```

- HAVING Filtering
  - Only filter rows that comply to having-clause

Single Table, no Index (1.6)

SELECT *
FROM emp
WHERE rowid=
  ‘00004F2A.00A2.000C’

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
```

- Table access by rowid
  - Single row lookup
  - Goes straight to the block, and filters the row
  - Fastest way to retreive one row
    - If you know its rowid

Single Table, Index (2.1)

```
SELECT *
FROM emp
WHERE empno=174;

Unique emp(empno)
```

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX unique scan i_emp_pk
```

- Index Unique Scan
  - Traverses the node blocks to locate correct leaf block
  - Searches value in leaf block (if not found => done)
  - Returns rowid to parent row-source
    - Parent: accesses the file+block and returns the row

Index Unique Scan (2.1)

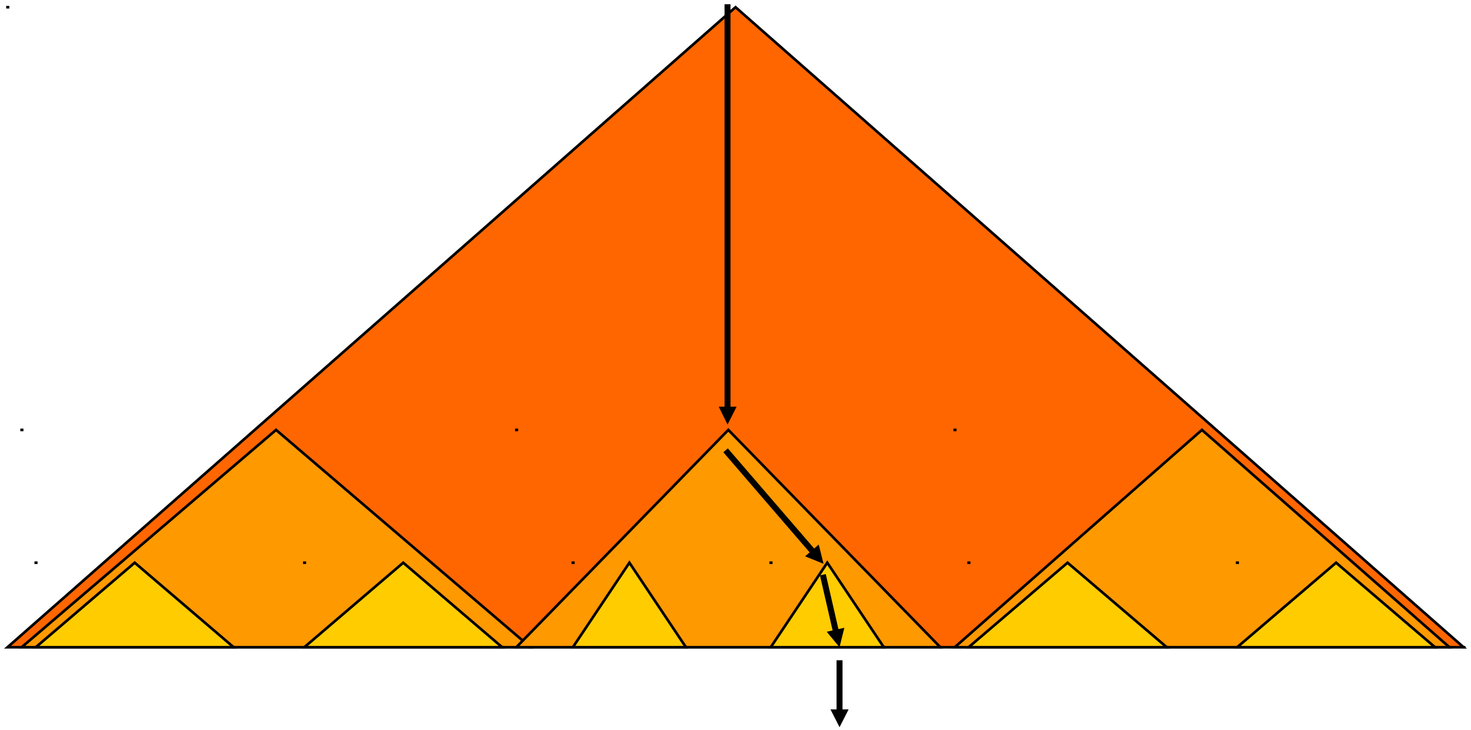


Table access by rowid

Single Table, Index (2.2)

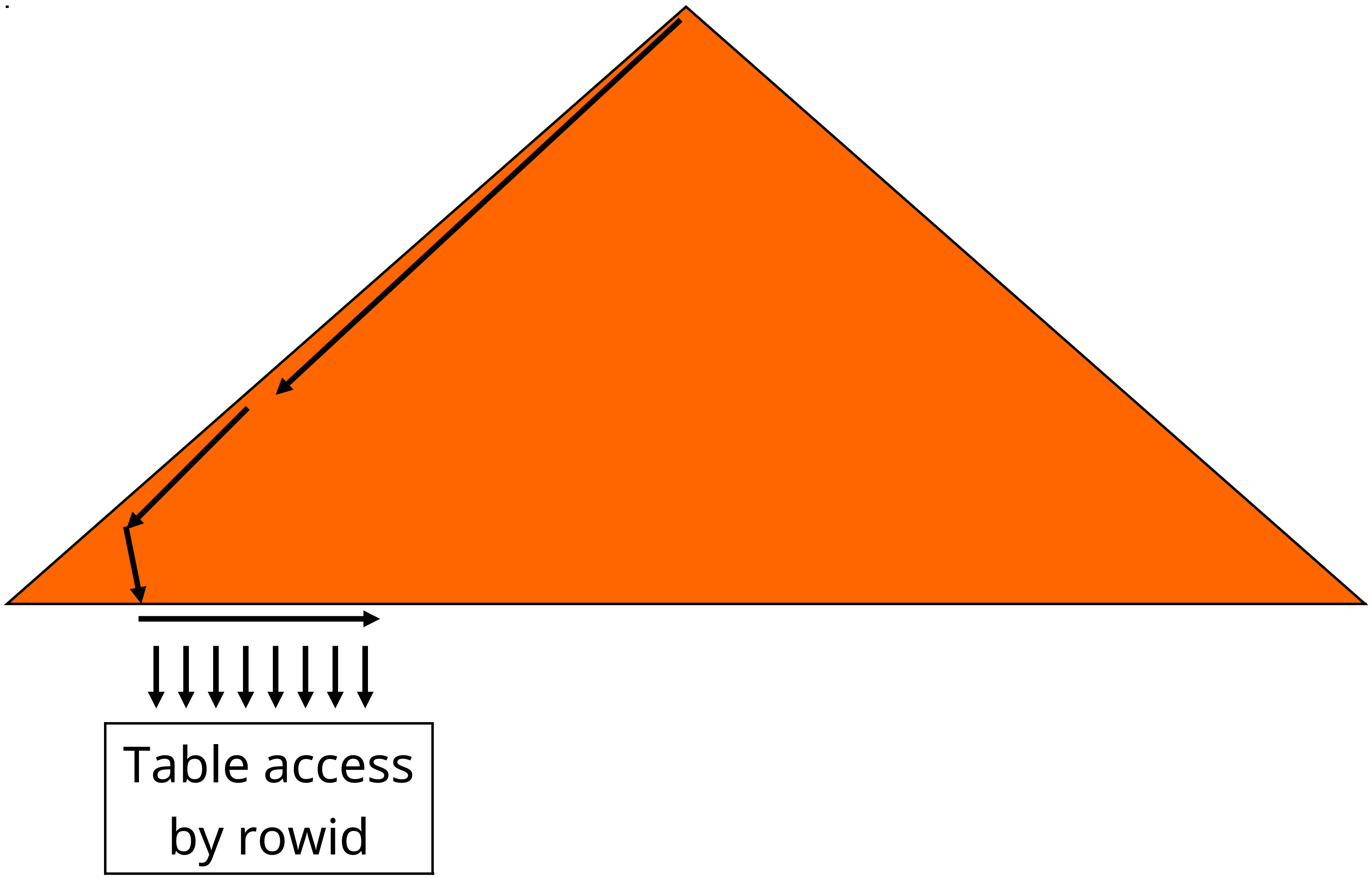
```
SELECT *
FROM emp
WHERE job='manager';


emp(job)
```

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_job
```

- (Non-unique) Index Range Scan
  - Traverses the node blocks to locate most left leaf block
  - Searches 1st occurrence of value in leaf block
  - Returns rowid to parent row-source
    - Parent: accesses the file+block and returns the row
  - Continues on to next occurrence of value in leaf block
    - Until no more occurences

Index Range Scan (2.2)



Single Table, Index (2.3)

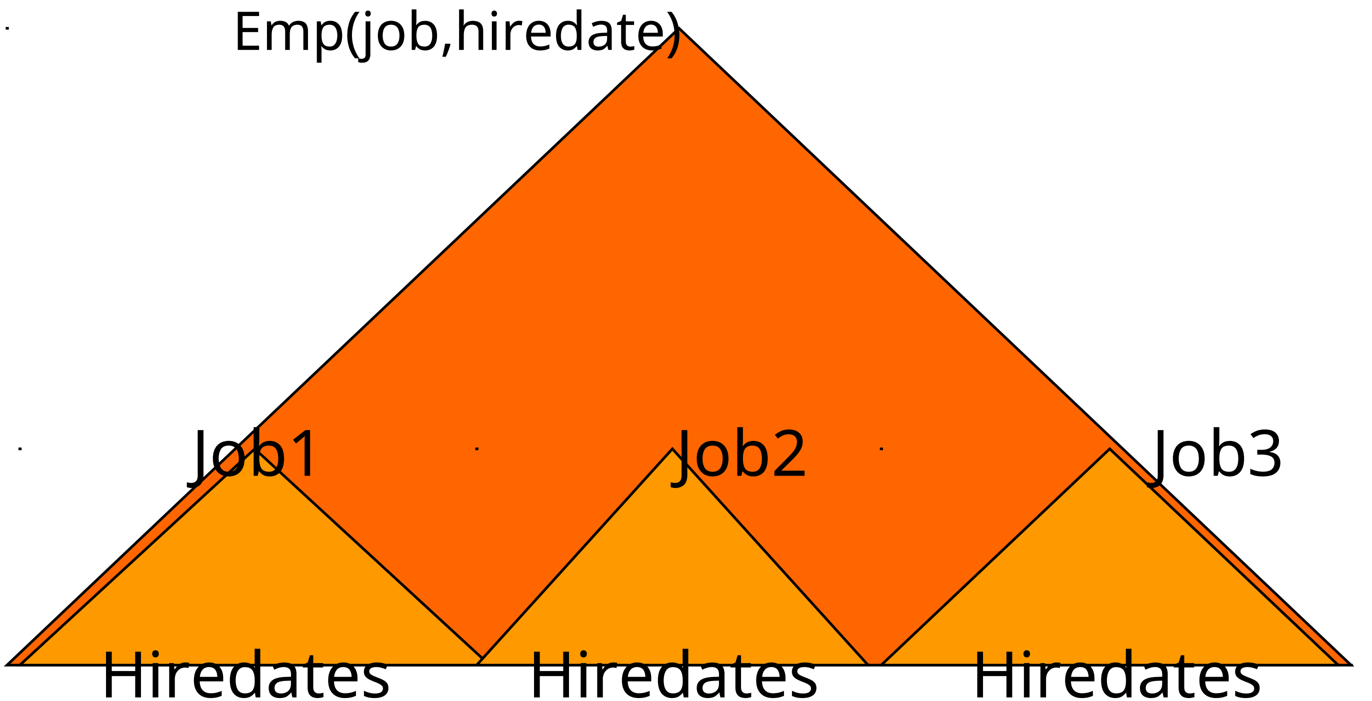
SELECT *
FROM emp
WHERE empno>100;

Unique emp(empno)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_pk
```

- Unique Index Range Scan
  - Traverses the node blocks to locate most left leaf block with start value
  - Searches 1st occurrence of value-range in leaf block
  - Returns rowid to parent row-source
    - Parent: accesses the file+block and returns the row
  - Continues on to next valid occurrence in leaf block
    - Until no more occurences / no longer in value-range

Concatenated Indexes



Multiple levels of Btrees, by column order

Single Table, Index (2.4)

SELECT *
FROM emp
WHERE job=‘manager’
AND hiredate=’01-01-2001’;

Emp(job,hiredate)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j_h
```

- Full Concatenated Index
  - Use job-value to navigate to sub-Btree
  - Then search all applicable hiredates

Single Table, Index (2.5)

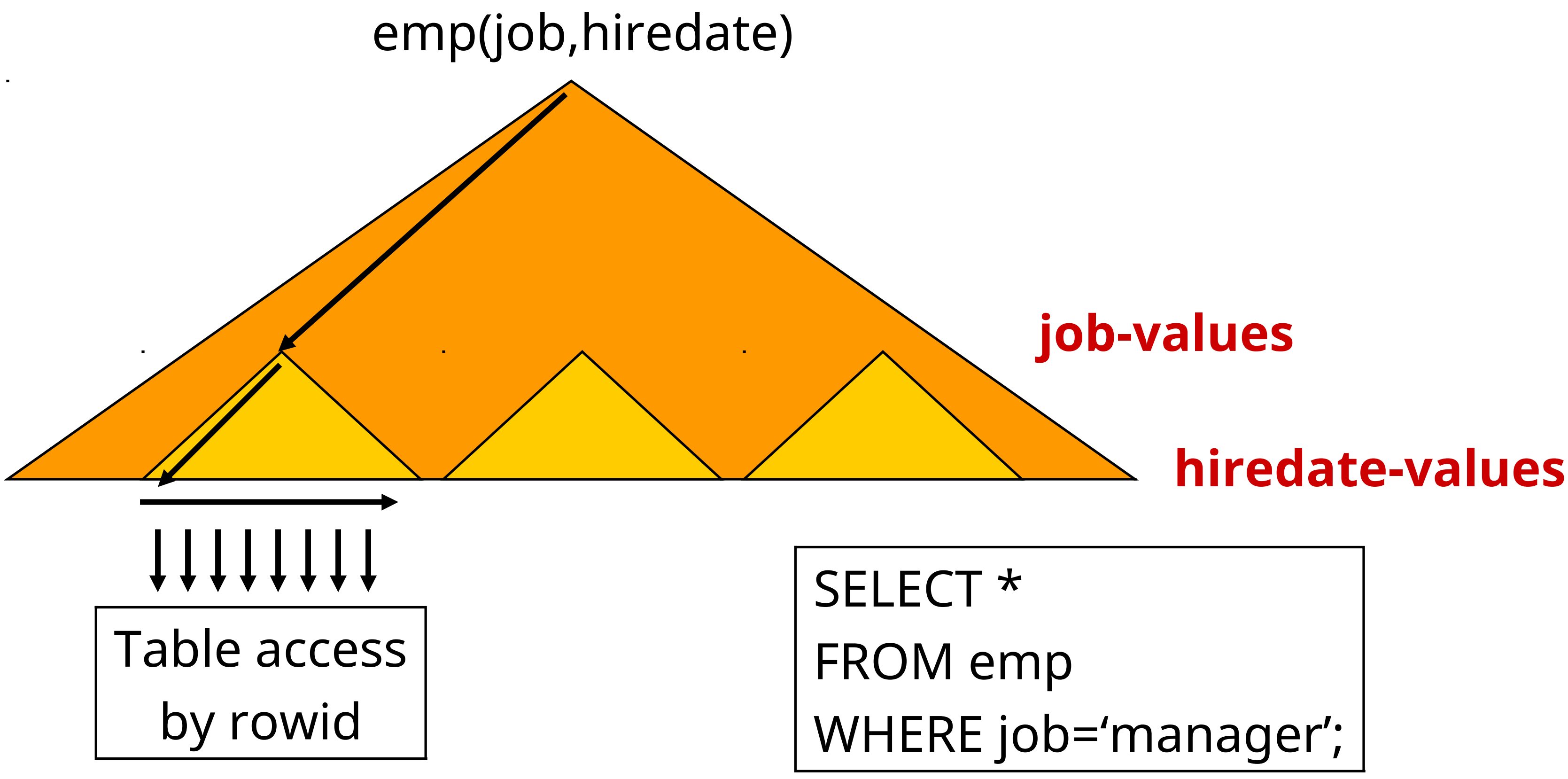
```
SELECT *
FROM emp
WHERE job='manager';

Emp(job,hiredate)
```

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j_h
```

- (Leading) Prefix of Concatenated Index
  - Scans full sub-Btree inside larger Btree

Index Range Scan (2.5)



Single Table, Index (2.6)

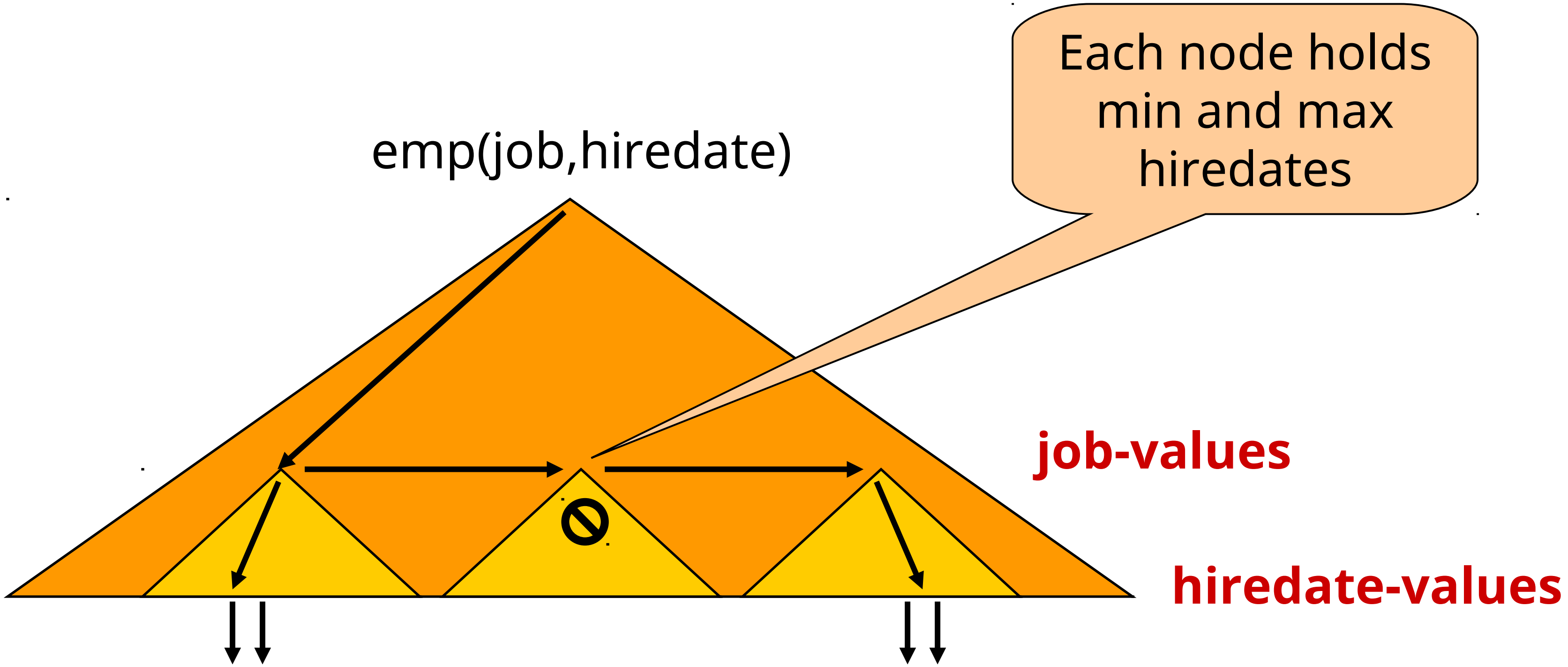
SELECT *
FROM emp
WHERE hiredate='01-01-2001';

Emp(job,hiredate)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j_h
```

- Index Skip Scan (prior versions did FTS)
  - "To use indexes where they've never been used before"
  - Predicate on leading column(s) no longer needed
  - Views Btree as collection of smaller sub-Btrees
  - Works best with low-cardinality leading column(s)

Index Skip Scan (2.6)




```
SELECT *
FROM emp
WHERE hiredate='01-01-2001';
```

Single Table, Index (2.7)

```
SELECT *
FROM emp
WHERE empno>100
AND job='manager';

Unique Emp(empno)
Emp(job)
```

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_job
```

- Multiple Indexes
  - Rule: uses heuristic decision list to choose which one
    - Avaliable indexes are 'ranked'
  - Cost: computes most selective one (ie. least costing)
    - Uses statistics

RBO Heuristics

- Ranking multiple available indexes
  1. Equality on single column unique index
  2. Equality on concatenated unique index
  3. Equality on concatenated index
  4. Equality on single column index
  5. Bounded range search in index
     - Like, Between, Leading-part, …
  6. Unbounded range search in index
     - Greater, Smaller (on leading part)

  Normally you hint which one to use

CBO Cost Computation

- Statistics at various levels
    - Table:
        - Num_rows, Blocks, Empty_blocks, Avg_space
    - Column:
        - Num_values, Low_value, High_value, Num_nulls
    - Index:
        - Distinct_keys, Blevel, Avg_leaf_blocks_per_key, Avg_data_blocks_per_key, Leaf_blocks
  - Used to compute selectivity of each index
    - Selectivity = percentage of rows returned
        - Number of I/O's plays big role
    - FTS is also considered at this time!

Single Table, Index (2.1)

SELECT *
FROM emp
WHERE empno=174;

Unique emp(empno)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX unique scan i_emp_pk
Or,
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

- CBO will use Full Table Scan If,
  # of I/O's to do FTS < # of I/O's to do IRS (Index Range Scan)
  - FTS I/O uses db_file_multiblock_read_count (dfmrc)
    - Typically 16
  - Unique scan uses: (blevel + 1) +1 I/O's
  - FTS uses ceil(#table blocks / dfmrc) I/O's

CBO: Clustering Factor

- Index level statistic
  - How well ordered are the rows in comparison to indexed values?
  - Average number of blocks to access a single value
    - **1** means range scans are cheap
    - **<# of table blocks>** means range scans are expensive
  - Used to rank multiple available range scans

```
Blck 1 Blck 2 Blck 3
------ ------ ------
 A A A  B B B  C C C
```

Clust.fact = 1

```
Blck 1 Blck 2 Blck 3
------ ------ ------
 A B C  A B C  A B C
```

Clust.fact = 3

Single Table, Index (2.2)

SELECT *
FROM emp
WHERE job='manager';

emp(job)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_job
Or,
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

- Clustering factor comparing IRS against FTS
  - If, (#table blocks / dfmrc)

    <

    (#values * clust.factor) + blevel + leafblocks-to-visit

    then, FTS is used

Single Table, Index (2.7)

SELECT *
FROM emp
WHERE empno>100
AND job='manager';

Unique Emp(empno)
Emp(job)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_job
Or,
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_empno
```

- Clust.factor comparing multiple IRS's
  - Suppose FTS is too many I/O's
  - Compare (#values * clust.fact) to decide which index
    - Empno-selectivity => #values * 1 => # I/O's
    - Job-selectivity => 1 * clust.fact => # I/O's

Single Table, Index (2.8)

```
SELECT *
FROM emp
WHERE job='manager'
AND depno=10

Emp(job)
Emp(depno)
```

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....AND-EQUAL
>.......INDEX range scan i_emp_job
>.......INDEX range scan i_emp_depno
```

- Multiple same-rank, single-column indexes
  - AND-EQUAL: merge up to 5 single column range scans
  - Combines multiple index range scans prior to table access
    - Intersects rowid sets from each range scan
  - Rarely seen with CBO

Single Table, Index (2.9)

SELECT ename

FROM emp

WHERE job=‘manager’;

Emp(job,ename)

```
>.SELECT STATEMENT
>...INDEX range scan i_emp_j_e
```

- Using indexes to avoid table access
  - Depending on columns used in SELECT-list and other places of WHERE-clause
  - No table-access if all used columns present in index

Single Table, Index (2.10)

```
SELECT count(*)
FROM big_emp;

Big_emp(empno)
```

```
>.SELECT STATEMENT
>...INDEX fast full scan i_emp_empno
```

- Fast Full Index Scan (CBO only)
  - Uses same multiblock I/O as FTS
  - Eligible index must have at least one NOT NULL column
  - Rows are returned leaf-block order
    - Not in indexed-columns-order

Joins, Nested Loops (3.1)

```
SELECT *
FROM dept, emp;
```

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....TABLE ACCESS full dept
>.....TABLE ACCESS full emp
```

- Full Cartesian Product via Nested Loop Join (NLJ)
  - Init(RowSource1);
    While not eof(RowSource1)
    Loop Init(RowSource2);
    While not eof(RowSource2)
    Loop return(CurRec(RowSource1)+CurRec(RowSource2));
    NxtRec(RowSource2);
    End Loop;
    NxtRec(RowSource1);
    End Loop;

Two loops, nested

Joins, Sort Merge (3.2)

```
SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#;
```

```
>.SELECT STATEMENT
>...MERGE JOIN
>.....SORT join
>.......TABLE ACCESS full emp
>.....SORT join
>.......TABLE ACCESS full dept
```

- Inner Join, no indexes: Sort Merge Join (SMJ)

```
Tmp1 := Sort(RowSource1,JoinColumn);
Tmp2 := Sort(RowSource2,JoinColumn);
Init(Tmp1); Init(Tmp2);
While Sync(Tmp1,Tmp2,JoinColumn)
Loop return(CurRec(Tmp1)+CurRec(Tmp2));
End Loop;
```

Sync advances pointer(s) to next match

Joins (3.3)

SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#;

Emp(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....TABLE ACCESS full dept
>.....TABLE ACCESS by rowid emp
>.......INDEX range scan e_emp_fk
```

- Inner Join, only one side indexed
  - NLJ starts with full scan of non-indexed table
  - Per row retrieved use index to find matching rows
    - Within 2nd loop a (current) value for d# is available!
    - And used to perform a range scan

Joins (3.4)

SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#

Emp(d#)
Unique Dept(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....TABLE ACCESS full dept
>.....TABLE ACCESS by rowid emp
>.......INDEX range scan e_emp_fk
Or,
>.SELECT STATEMENT
>...NESTED LOOPS
>.....TABLE ACCESS full emp
>.....TABLE ACCESS by rowid dept
>.......INDEX unique scan e_dept_pk
```

- Inner Join, both sides indexed
  - RBO: NLJ, start with FTS of last table in FROM-clause
  - CBO: NLJ, start with FTS of biggest table in FROM-clause
    - Best multi-block I/O benefit in FTS
    - More likely smaller table will be in buffer cache

Joins (3.5)

SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#
AND dept.loc = 'DALLAS'

Emp(d#)
Unique Dept(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....TABLE ACCESS full dept
>.....TABLE ACCESS by rowid emp
>.......INDEX range scan e_emp_fk
```

- Inner Join with additional conditions
  - Nested Loops
  - Always starts with table thas has extra condition(s)

Hashing

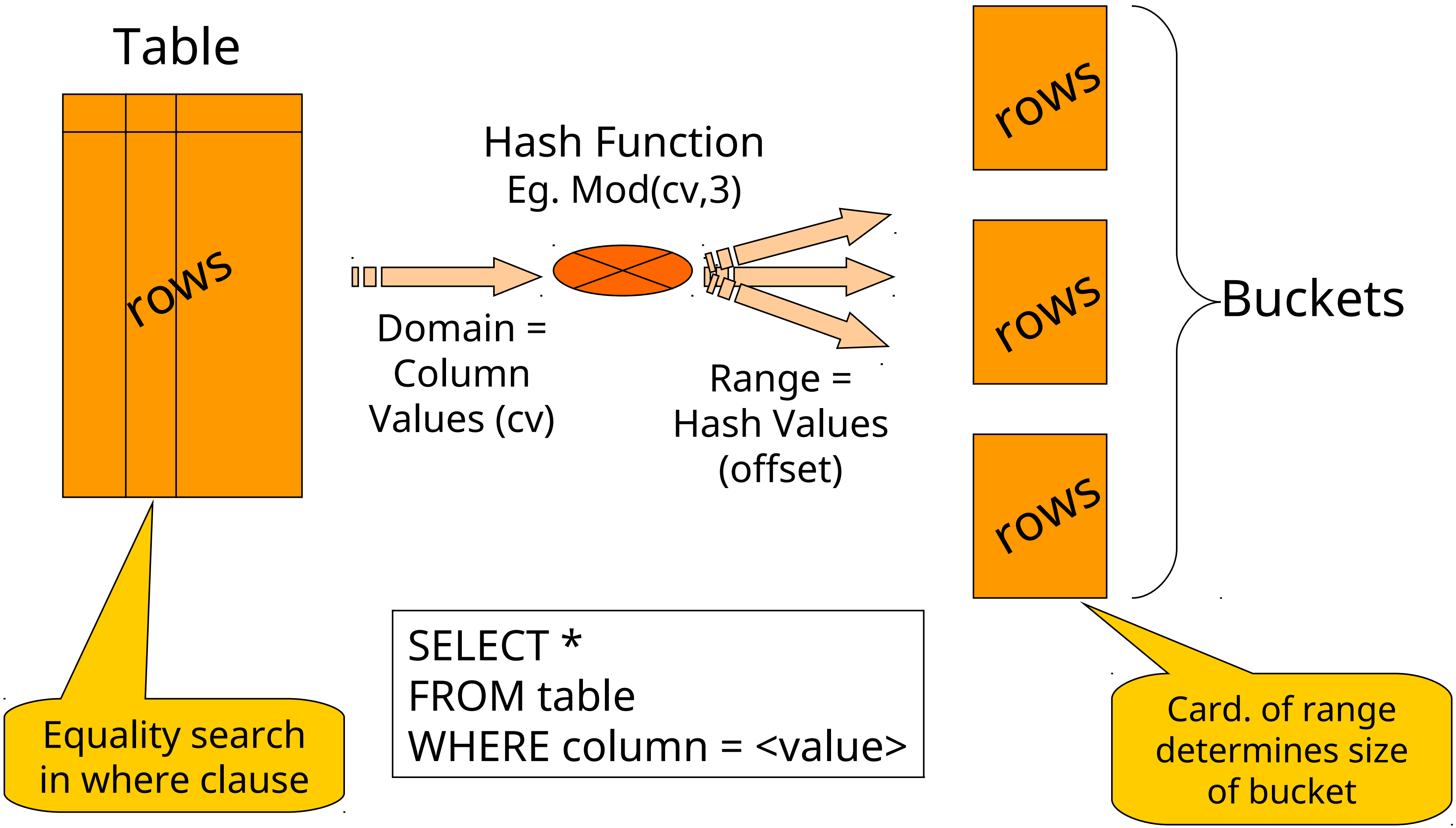


Table
rows
Hash Function
Eg. Mod(cv,3)
Domain =
Column
Values (cv)
Range =
Hash Values
(offset)
rows
rows
rows
Buckets
Equality search
in where clause
SELECT *
FROM table
WHERE column = <value>
Card. of range
determines size
of bucket

Joins, Hash (3.6)

```
SELECT *
FROM dept, emp
WHERE dept.d# = emp.d#

Emp(d#), Unique Dept(d#)
```

```
>.SELECT STATEMENT
>...HASH JOIN
>.....TABLE ACCESS full dept
>.....TABLE ACCESS full emp
```

```
– Tmp1 := Hash(RowSource1,JoinColumn);        -- In memory
  Init(RowSource2);
  While not eof(RowSource2)
  Loop HashInit(Tmp1,JoinValue);              -- Locate bucket
        While not eof(Tmp1)
        Loop return(CurRec(RowSource2)+CurRec(Tmp1));
              NxtHashRec(Tmp1,JoinValue);
        End Loop; NxtRec(RowSource2);
  End Loop;
```

Joins, Hash (3.6)

- Must be explicitely enabled via init.ora file:
  - Hash_Join_Enabled = True
  - Hash_Area_Size = <bytes>
- If hashed table does not fit in memory
  - 1st rowsource: temporary hash cluster is built
    - And written to disk (I/O's) in partitions
  - 2nd rowsource also converted using same hash-function
  - Per 'bucket' rows are matched and returned
    - One bucket must fit in memory, else very bad performance

Subquery (4.1)

```
SELECT dname, deptno
FROM dept
WHERE d# IN
  (SELECT d#
   FROM emp);
```

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....VIEW
>.......SORT unique
>.........TABLE ACCESS full emp
>.....TABLE ACCESS by rowid dept
>.......INDEX unique scan i_dept_pk
```

- Transformation into join
  - Temporary view is built which drives the nested loop

Subquery, Correlated (4.2)

```
SELECT *
FROM emp e
WHERE sal >
  (SELECT sal
   FROM emp m
   WHERE m.e#=e.mgr#)
```

```
>.SELECT STATEMENT
>...FILTER
>.....TABLE ACCESS full emp
>.....TABLE ACCESS by rowid emp
>.......INDEX unique scan i_emp_pk
```

- "Nested Loops"-like FILTER
  - For each row of 1st rowsource, execute 2nd rowsource and filter on truth of subquery-condition
  - Subquery can be re-written as self-join of EMP table

Subquery, Correlated (4.2)

```
SELECT *
FROM emp e, emp m
WHERE m.e#=e.mgr#
AND e.sal > m.sal;
```

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....TABLE ACCESS full emp
>.....TABLE ACCESS by rowid emp
>.......INDEX unique scan i_emp_pk
```

- Subquery rewrite to join
  - Subquery can also be rewritten to EXISTS-subquery

Subquery, Correlated (4.2)

```
SELECT *
FROM emp e
WHERE exists
 (SELECT 'less salary'
  FROM emp m
  WHERE e.mgr# = m.e#
            and m.sal < e.sal);
```

```
>.SELECT STATEMENT
>...FILTER
>.....TABLE ACCESS full emp
>.....TABLE ACCESS by rowid emp
>.......INDEX unique scan i_emp_pk
```

- Subquery rewrite to EXISTS query
  - For each row of 1st rowsource, execute 2nd rowsource And filter on retrieval of rows by 2nd rowsource

Concatenation (4.3)

SELECT *
FROM emp
WHERE mgr# = 100
OR job = ‘CLERK’;

Emp(mgr#)
Emp(job)

```
>.SELECT STATEMENT
>...CONCATENATION
>.....TABLE ACCESS by rowid emp
>.......INDEX range scan i_emp_m
>.....TABLE ACCESS by rowid emp
>.......INDEX range scan i_emp_j
```

- Concatenation (OR-processing)
  - Similar to query rewrite into 2 seperate queries
  - Which are then ‘concatenated’
  - If one index was missing => Full Table Scan

Inlist Iterator (4.4)

SELECT *
FROM dept
WHERE d# in (10,20,30);

Unique Dept(d#)

```
>.SELECT STATEMENT
>...INLIST ITERATOR
>.....TABLE ACCESS by rowid dept
>.......INDEX unique scan i_dept_pk
```

- Iteration over enumerated value-list
  - Every value executed seperately
- Same as concatenation of 3 “OR-red” values

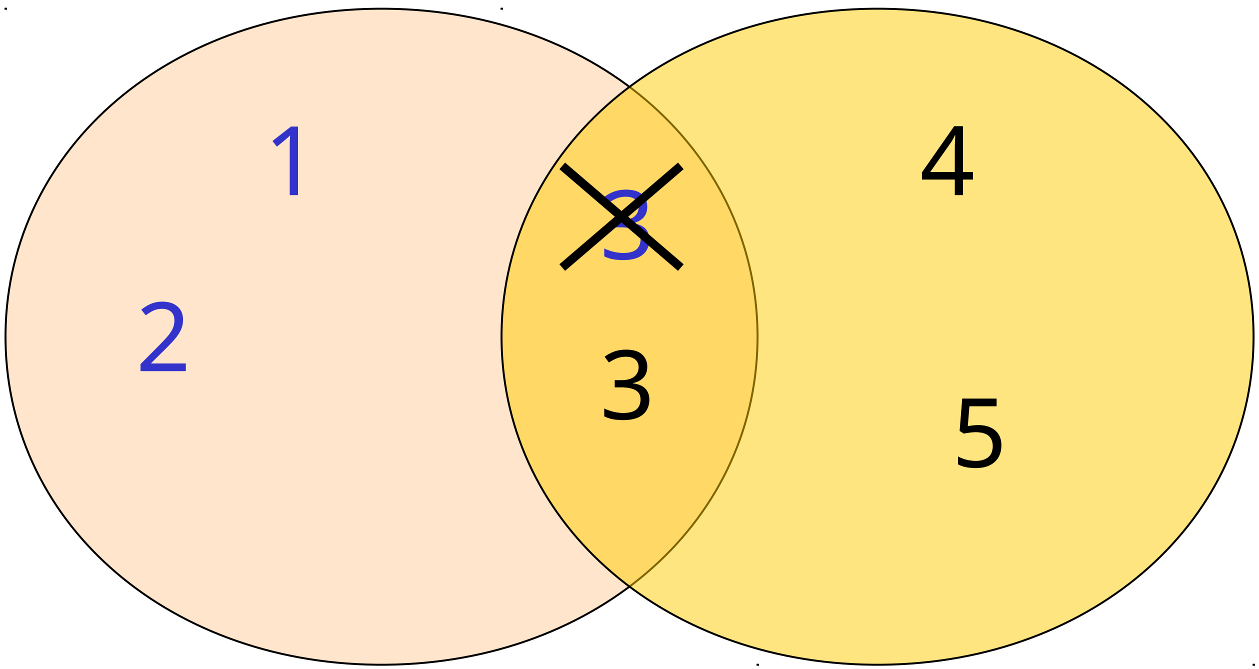
Union (4.5)

```
SELECT empno
FROM emp
UNION
SELECT deptno
FROM dept;
```

```
>.SELECT STATEMENT
>...SORT unique
>.....UNION
>.......TABLE ACCESS full emp
>.......TABLE ACCESS full dept
```

- Union followed by Sort-Unique
  - Sub rowsources are all executed/optimized individually
  - Rows retrieved are ‘concatenated’
  - Set theory demands unique elements (Sort)

UNION



Union All (4.6)

```
SELECT empno
FROM emp
UNION ALL
SELECT deptno
FROM dept;
```

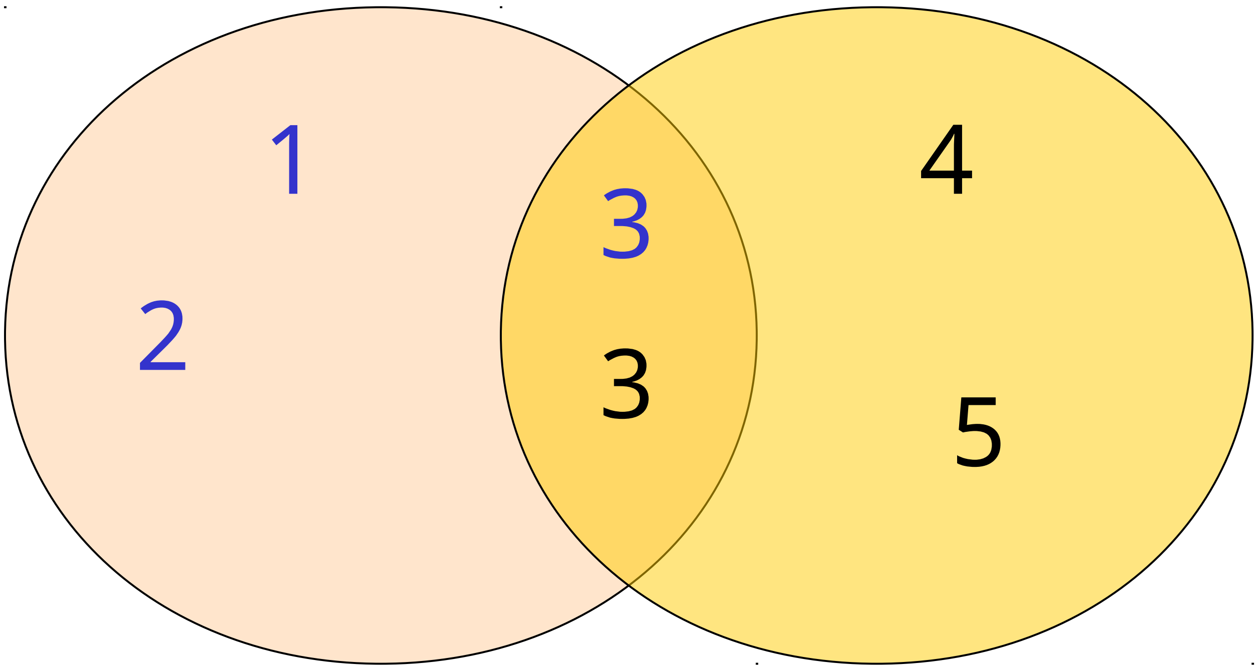
```
>.SELECT STATEMENT
>...UNION-ALL
>.....TABLE ACCESS full emp
>.....TABLE ACCESS full dept
```

- Union-All: result is a ‘bag’, not a set
  - (expensive) Sort-operator not necessary

Use UNION-ALL if you know the bag is a set.

(saving an expensive sort)

UNION ALL



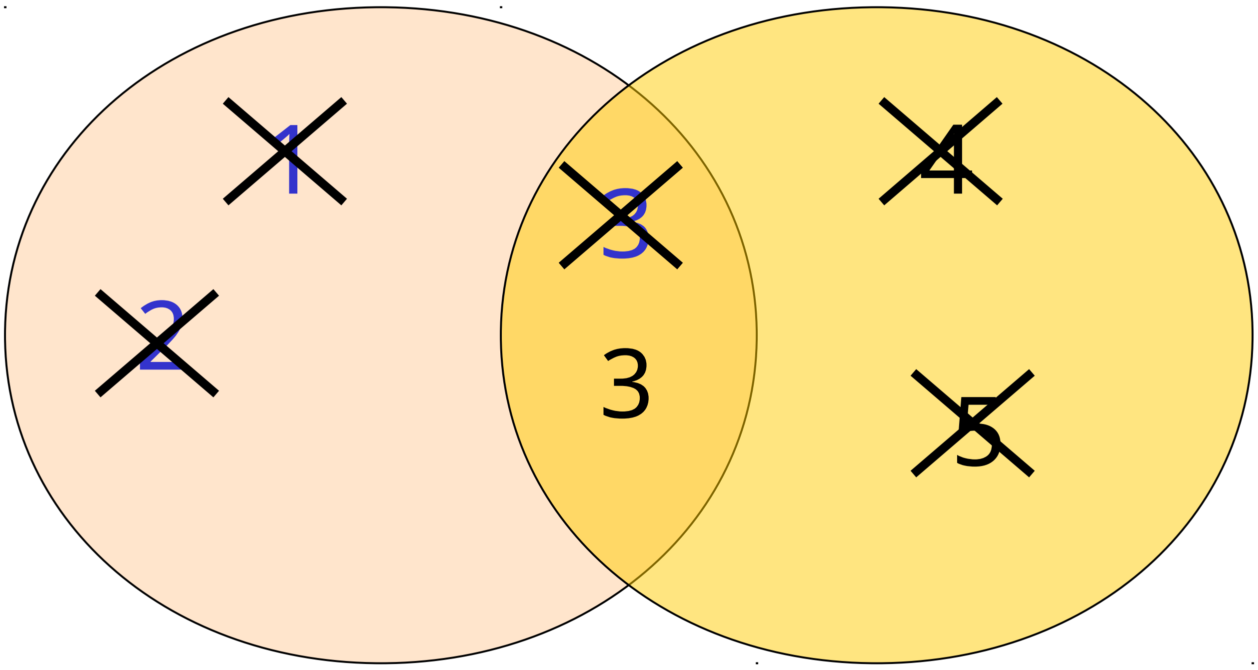
Intersect (4.7)

```sql
SELECT empno
FROM emp
INTERSECT
SELECT deptno
FROM dept;
```

```
>.SELECT STATEMENT
>...INTERSECTION
>.....SORT unique
>.......TABLE ACCESS full emp
>.....SORT unique
>.......TABLE ACCESS full dept
```

- INTERSECT
  - Sub rowsources are all executed/optimized individually
  - Very similar to Sort-Merge-Join processing
  - Full rows are sorted and matched

INTERSECT



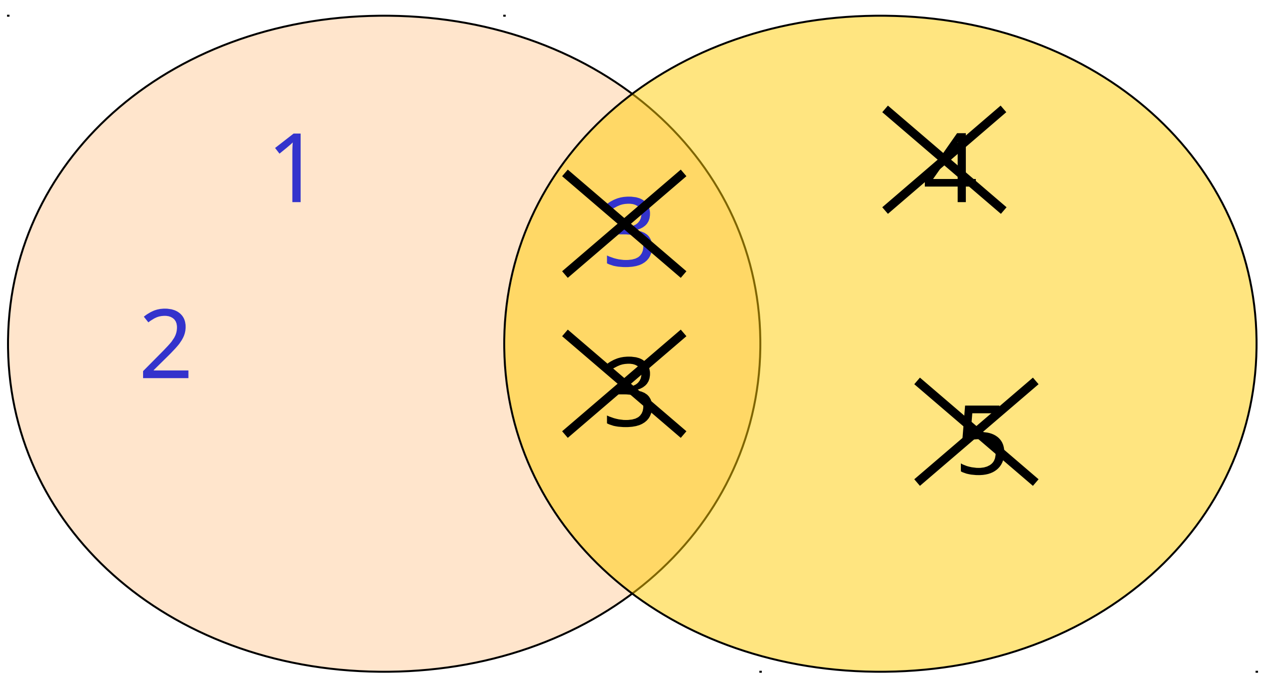
Minus (4.8)

```
SELECT empno
FROM emp
MINUS
SELECT deptno
FROM dept;
```

```
>.SELECT STATEMENT
>...MINUS
>.....SORT unique
>.......TABLE ACCESS full emp
>.....SORT unique
>.......TABLE ACCESS full dept
```

- MINUS
  - Sub rowsources are all executed/optimized individually
  - Similar to INTERSECT processing
    - Instead of match-and-return, match-and-exclude

MINUS



Utilities

- Tracing
- SQL Hints
- Analyze command
- Dbms_Stats package

Trace Files

- Explain-plan: give insight before execution
- Tracing: give insight in actual execution
    - CPU-time spent
    - Elapsed-time
    - # of physical block-I/O's
    - # of cached block-I/O's
    - Rows-processed per row-source
- Session must be put in trace-mode
    - Alter session set sql_trace=true;
    - Exec dbms_system.set_sql_trace_in_session(sid,s#,T/F);

Trace Files

- Tracefile is generated on database server
  - Needs to be formatted with TKPROF-utility

    tkprof <tracefile> <tkp-file> <un>/<pw>

  - Two sections per SQL-statement:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.06       0.07          0          0          0           0
Execute      1      0.01       0.01          0          0          0           0
Fetch        1      0.11       0.13          0         37          2           2
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.18       0.21          0         37          2           2
```

Trace Files

- 2nd section: extended explain plan:
  - Example 4.2 (emp with more sal than mgr),

```
#R  Plan                                                        .
 2  SELECT STATEMENT
14    FILTER
14      TABLE ACCESS (FULL) OF 'EMP'
11      TABLE ACCESS (BY ROWID) OF 'EMP'
12        INDEX (UNIQUE SCAN) OF 'I_EMP_PK' (UNIQUE)
```

- Emp has 14 records
- Two of them have no manager (NULL mgr column value)
- One of them points to non-existing employee
- Two actually earn more than their manager

Hints

- Force optimizer to pick specific alternative
  - Implemented via embedded comment

```
SELECT /*+ <hint> */ ....
FROM ....
WHERE ....

UPDATE /*+ <hint> */ ....
WHERE ....

DELETE /*+ <hint> */ ....
WHERE ....

INSERT (see SELECT)
```

Hints

- Common hints
  - Full(<tab>)
  - Index(<tab> <ind>)
  - Index_asc(<tab> <ind>)
  - Index_desc(<tab> <ind>)
  - Ordered
  - Use_NL(<tab> <tab>)
  - Use_Merge(<tab> <tab>)
  - Use_Hash(<tab> <tab>)
  - Leading(<tab>)
  - First_rows, All_rows, Rule

Analyze command

- Statistics need to be periodically generated
  - Done via ‘ANALYZE’ command

    Analyze <Table | Index> <x>
    <compute | estimate | delete> statistics
    <sample <x> <Rows | Percent>>

    Analyze table emp estimate statistics sample 30 percent;

ANALYZE will be de-supported

Dbms_Stats Package

- Successor of Analyze command

  - Dbms_stats.gather_index_stats(<owner>,<index>, <blocksample>,<est.percent>)
  - Dbms_stats.gather_table_stats(<owner>,<table>, <blocksample>,<est.percent>)
  - Dbms_stats.delete_index_stats(<owner>,<index>)
  - Dbms_stats.delete_table_stats(<owner>,<table>)

SQL>exec dbms_stats.gather_table_status(‘scott’,’emp’,null,30);

Warehouse Specifics

- Traditional Star Query
- Bitmap Indexes
  - Bitmap merge, and, conversion-to-rowid
  - Single table query
- Star Queries
  - Multiple tables

Traditional Star Query

SELECT f.*
FROM a,b,f
WHERE a.pk = f.a_fk
AND b.pk = f.b_fk
AND a.t = … AND b.s = …

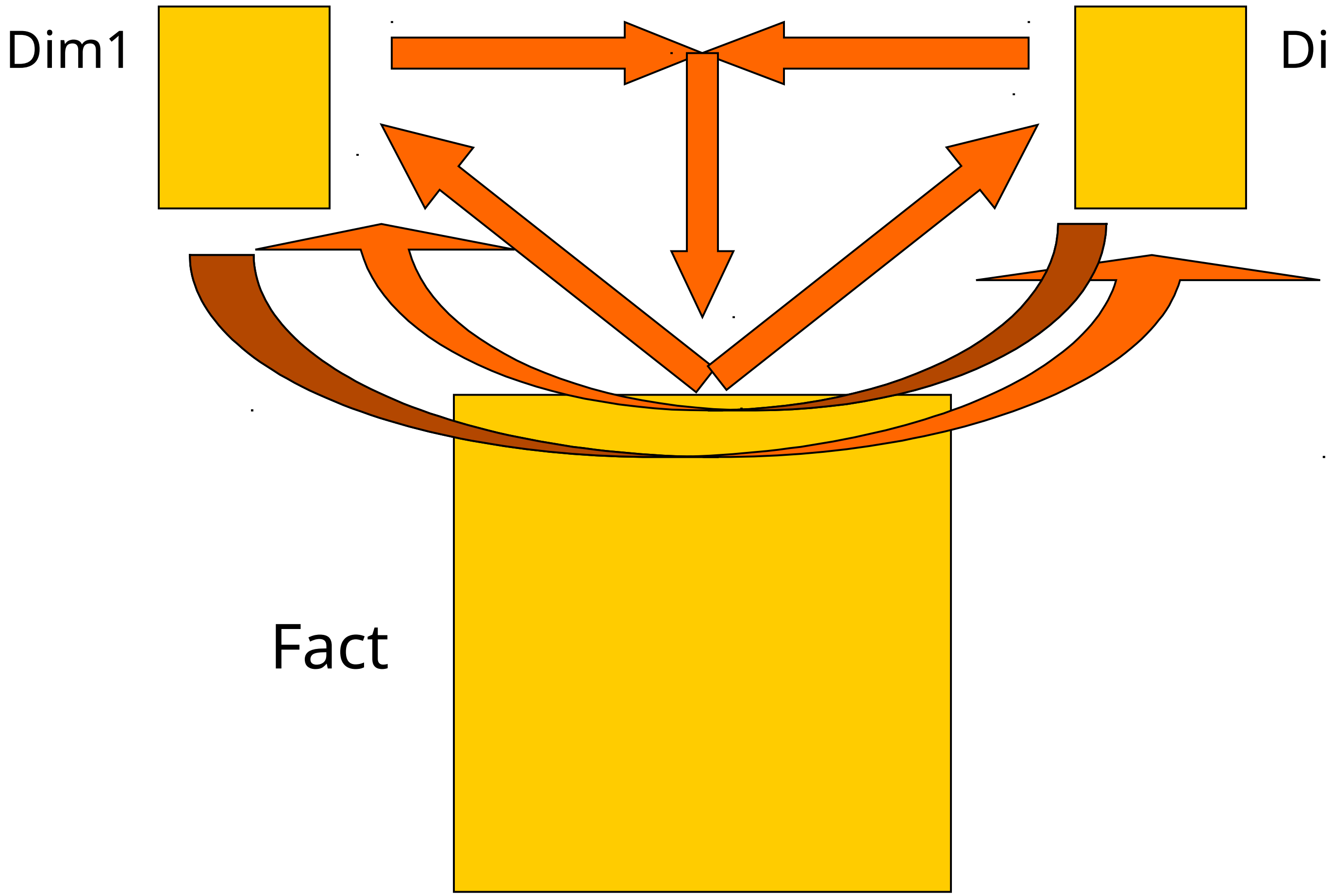
A(pk), B(pk)
F(a_fk), F(b_fk)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....NESTED LOOPS
>.......TABLE ACCESS full b
>.......TABLE ACCESS by rowid fact
>.........INDEX range scan i_fact_b
>.....TABLE ACCESS by rowid a
>.......INDEX unique scan a_pk
```

- Double nested loops
  - Pick one table as start (A or B)
  - Then follow join-conditions using Nested_Loops

Too complex for AND-EQUAL

Traditional Star Query



Four access-order alternatives!

Traditional Star Query

```
SELECT f.*
FROM a,b,f
WHERE a.pk = f.a_fk
AND b.pk = f.b_fk
AND a.t = …  AND b.s = …

F(a_fk,b_fk,…)
```

```
>.SELECT STATEMENT
>...NESTED LOOPS
>.....MERGE JOIN cartesian
>.......TABLE ACCESS full a
>.......SORT join
>.........TABLE ACCESS full b
>.....TABLE ACCESS by rowid fact
>.......INDEX range scan I_f_abc
```

- Concatenated Index Range Scans for Star Query
  - At least two dimensions
  - Index at least one column more than dimensions used
  - Merge-Join-Cartesian gives all applicable dimension combinations
  - Per combination the concatenated index is probed

Bitmap Access, Single Table

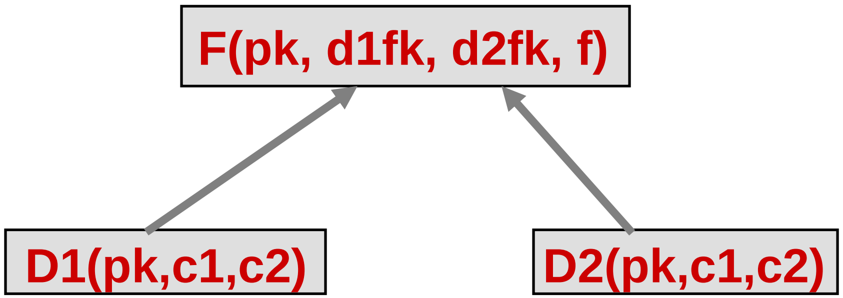
```
SELECT count(*)
FROM customer
WHERE status='M'
AND region in ('C','W');
```

```
>.......TABLE ACCESS (BY INDEX ROWID) cust
>.........BITMAP CONVERSION to rowids
>...........BITMAP AND
>.............BITMAP INDEX single value cs
>.............BITMAP MERGE
>...............BITMAP KEY ITERATION
>.................BITMAP INDEX range scan cr
```

- Bitmap OR's, AND's and CONVERSION
  - Find Central and West bitstreams (bitmap key-iteration)
  - Perform logical OR on them (bitmap merge)
  - Find Married bitstream
  - Perform logical AND on region bitstream (bitmap and)
  - Convert to actual rowid's
  - Access table

Bitmap Access, Star Query

Bitmap indexes: id1, id2




```
SELECT sum(f)
FROM F,D1,D2
WHERE F=D1 and F=D2
AND D1.C1=<…>
AND D2.C2=<…>
```

```
>.......TABLE ACCESS (BY INDEX ROWID) f
>.........BITMAP CONVERSION (TO ROWIDS)
>...........BITMAP AND
>.............BITMAP MERGE
>...............BITMAP KEY ITERATION
>.................TABLE ACCESS (FULL) d1
>.................BITMAP INDEX (RANGE SCAN) id1
>.............BITMAP MERGE
>...............BITMAP KEY ITERATION
>.................TABLE ACCESS (FULL) d2
>.................BITMAP INDEX (RANGE SCAN) id2
```

Warehouse Hints

- Specific star-query related hints
  - Star
    - Traditional: via concat-index range scan
  - Star_transformation
    - Via single column bitmap index merges/and's
  - Fact(t) / No_fact(t)
    - Help star_transformation
  - Index_combine(t i1 i2 ...)
    - Explicitely instruct which indexes to merge/and

SQL Tuning: Roadmap

- Able to read plan
- Able to translate plan into 3GL program
    - Know your row-source operators
- Able to read SQL
- Able to translate SQL into business query
    - Know your datamodel
- Able to judge outcome
    - Know your business rules / data-statistics
        - Better than CBO does
- Experts:
    - Optimize SQL while writing SQL…