Db2 Query Optimization 101

John Hornibrook
IBM Canada

Db2 LUW

Optimal query access plans are essential for good data server performance and it is the Db2 query optimizer's job to choose the best access plan. The optimizer is a very sophisticated component of the data server, tasked with the challenging job of choosing good access paths for the variety of features and table organizations supported by Db2. The optimizer can automatically rewrite complex SQL resulting in huge performance improvements. It models various aspects of Db2 runtime so that it can choose the best access plan out of hundreds of thousands of possible options. Attend this session to get an overview of how the optimizer works and to get some tips on how to understand its decisions and control its behavior.

Agenda

- What is query optimization and why is it important for performance?
- The different phases of query optimization
- How catalog statistics are used in query optimization
- How the query optimizer costs access plans
- Understand access plans using the explain facility

2

Why Optimize Queries (1|2)?

- Performance
  - Improvement can be orders of magnitude for complex queries

- Lower total cost of ownership
  - Query tuning requires deep skill
    - Complex DB designs
    - SQL/XQuery generated by query generators, naive users
    - Fewer skilled administrators available
    - Various configuration and physical implementation

3

This Photo by Unknown Author is licensed under CC BY-SA

Why Optimize Queries (2|2)?

- There are a lot of factors to consider when optimizing query execution:
  - Configuration options
    - Memory, CPUs, I/O, communication channels
  - Table organization schemes
    - DB partitioning, table partitioning, multi-dimensional clustering
  - Data formats
    - Column, row, Hadoop
  - Complex data types
    - XML
  - Federation, data virtualization
    - Parts of the query execute on remote DB servers.
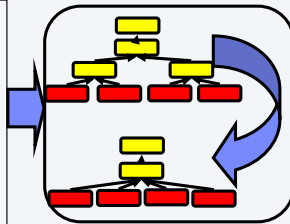 - Auxiliary performance and storage options
    - Indexes, MQTs, compression

4
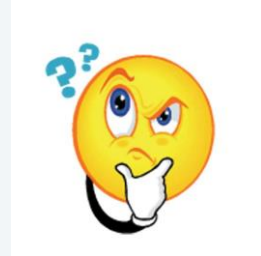
What is Query Optimization?

- SQL compilation:
  - In: <u>SQL statement</u>, Out: <u>access section</u>
  - Query optimization is 2 steps in the Db2 SQL statement compilation process
    - Query transformation (rewrite)
    - Access plan generation

Access section

Query transformation
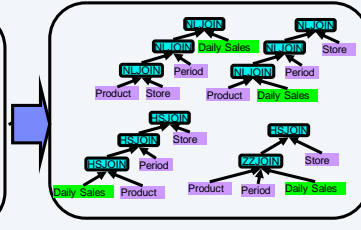
Access plan generation

```
SELECT ITEM_DESC, SUM(QUANTITY_SOLD),
     AVG(PRICE), AVG(COST)
  FROM PERIOD, DAILY_SALES, PRODUCT, STORE
WHERE
  PERIOD.PERKEY=DAILY_SALES.PERKEY AND
  PRODUCT.PRODKEY=DAILY_SALES.PRODKEY
  AND
  STORE.STOREKEY=DAILY_SALES.STOREKEY AND
  CALENDAR_DATE BETWEEN AND
  '01/01/2012' AND '04/28/2012' AND
  STORE_NUMBER='03' AND
  CATEGORY=72
GROUP BY ITEM_DESC
```



```
Thread 0
DSS
TQA (tq1)
AGG (complete)
BNO
EXT
Thread 1
TA (Product)
NLJN (Daily Sales)
NLJN (Period)
NLJN (Store)
AGG (partial)
TQB (tq1)
EXT
Thread 2
TA (DS_IX7)
EXT
Thread 3
TA (PER_IX2)
EXT
Thread 4
TA (ST_IX1)
EXT
```

Dozens of query transformations

Hundreds or thousands of access plan options

5

Phases of SQL Compilation
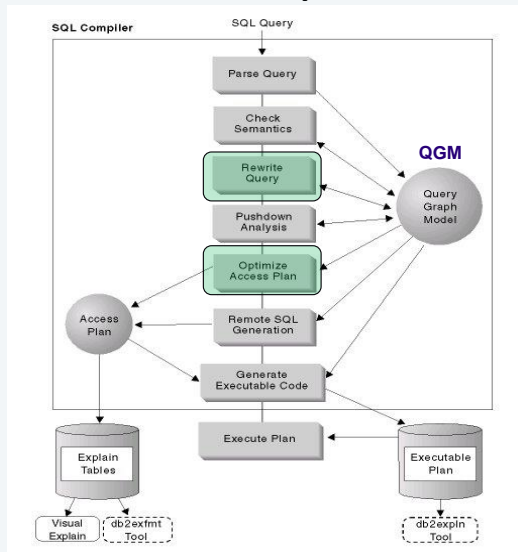
- Sometimes references to "**optimization**" really mean **SQL compilation**
- There is a lot more involved to SQL compilation

Parsing
- Catch syntax errors
- Generate internal representation of query

Semantic checking
- Determine if query makes sense
- Incorporate view definitions
- Add logic for constraint checking and triggers

Query optimization
- **Modify query to improve performance (Query Rewrite)**
- **Choose the most efficient "access plan"**

Pushdown Analysis
- Federation "optimization"

Threaded code generation
- Generate efficient "executable" code
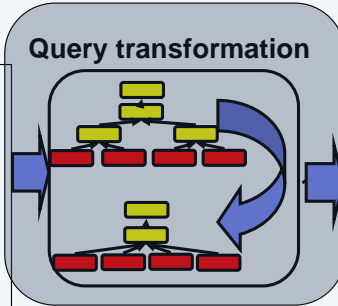- "Access section"

6

Query Optimization

- SQL compilation:
  - **Query transformation (rewrite)**
  - Access plan generation

Access section

```
SELECT ITEM_DESC, SUM(QUANTITY_SOLD),
    AVG(PRICE), AVG(COST)
  FROM PERIOD, DAILY_SALES, PRODUCT, STORE
WHERE
    PERIOD.PERKEY=DAILY_SALES.PERKEY AND
    PRODUCT.PRODKEY=DAILY_SALES.PRODKEY
     AND
    STORE.STOREKEY=DAILY_SALES.STOREKEY AND
    CALENDAR_DATE BETWEEN AND
     '01/01/2012' AND '04/28/2012' AND
    STORE_NUMBER='03' AND
    CATEGORY=72
GROUP BY ITEM_DESC
```

Query transformation

Access plan generation

Dozens of query transformations

Hundreds or thousands of access plan options

```
Thread 0
DSS
TQA (tq1)
AGG (complete)
BNO
EXT
Thread 1
TA (Product)
NLJN (Daily Sales)
NLJN (Period)
NLJN (Store)
AGG (partial)
TQB (tq1)
EXT
Thread 2
TA (DS_IX7)
EXT
Thread 3
TA (PER_IX2)
EXT
Thread 4
TA (ST_IX1)
EXT
```
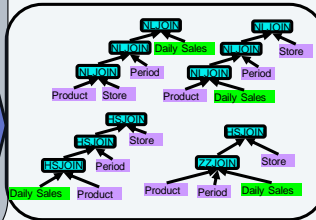
7

Query Rewrite - An Overview

- What is Query Rewrite?
  - Rewriting a given SQL query into a semantically equivalent form that may be processed more efficiently
- Example:
  - Original query:
    ```
    SELECT DISTINCT CUSTKEY, NAME FROM CUSTOMER
    ```
  - After Query Rewrite:
    ```
    SELECT CUSTKEY, NAME FROM CUSTOMER
    ```
  - Rationale:
    - CUSTKEY is unique, distinct is redundant

8

Query Rewrite - Why?

- Hidden culprit:
  - Multiple specifications allowed in SQL
  - SQL allows multiple specifications ;-)
  - There are many ways to express the same query
- Visible reasons:
  - Query generators
    - Often produce suboptimal queries that don't perform well
    - Don't permit "hand optimization"
  - Complex queries
    - Often result in redundancy, especially with views
  - Large data volumes
    - Optimal access plans more crucial
    - Penalty for poor planning is greater

9

Let's follow an example

"**Get the total and average shipping cost for NY catalog sales that had no returns for the 60 days starting Apr. 1 2018**"

```
SELECT
  SUM(CS_EXT_SHIP_COST) AS "TOTAL SHIPPING COST",
  AVG(CS_EXT_SHIP_COST) AS "AVERAGE SHIPPING COST"
FROM
  CATALOG_SALES CS1,
  DATE_DIM,
  CUSTOMER_ADDRESS
WHERE
  D_DATE BETWEEN '2018-4-01' AND (CAST('2018-4-01' AS DATE) + 60 DAYS) AND
  CS1.CS_SHIP_DATE_SK = D_DATE_SK AND
  CS1.CS_SHIP_ADDR_SK = CA_ADDRESS_SK AND
  CA_STATE = 'NY' AND
  NOT EXISTS
  (SELECT * FROM CATALOG_RETURNS CR1 WHERE CS1.CS_ORDER_NUMBER = CR1.CR_ORDER_NUMBER )
```

3 tables
(2 joins)

Search conditions
(predicates)

10

Step 1: Parsing and Query Graph Construction

- An SQL statement is parsed into a *Query Graph*
- Yellow boxes are relational operations
- Red boxes are tables or table functions

SELECT 1

GROUP BY

SUM(CS_EXT_SHIP_COST),
AVG(CS_EXT_SHIP_COST

SELECT 2

D_DATE BETWEEN '2018-4-01' AND
(CAST('2018-4-01' AS DATE) + 60 DAYS) AND
CS1.CS_SHIP_DATE_SK = D_DATE_SK AND
CS1.CS_SHIP_ADDR_SK = CA_ADDRESS_SK AND
CA_STATE = 'NY' AND NOT EXISTS (SELECT 3)

Correlation!

DATE_DIM   CATALOG SALES   CUSTOMER ADDRESS   SELECT 3

SELECT * FROM
CATALOG_RETURNS CR1
WHERE
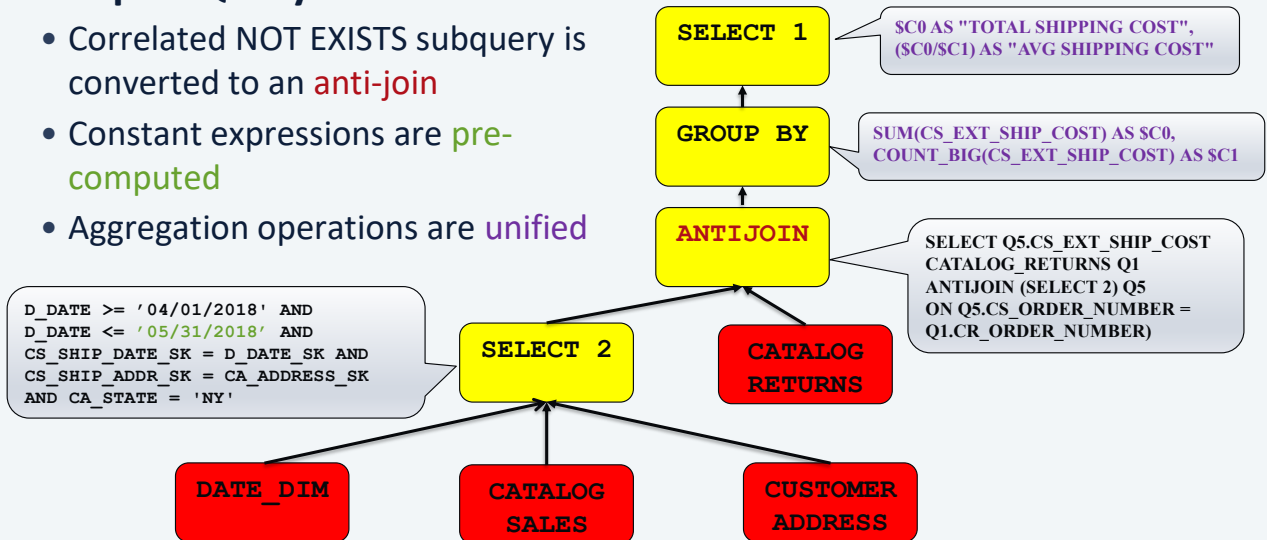CS1.CS_ORDER_NUMBER =
CR1.CR_ORDER_NUMBER

CATALOG RETURNS

The SQL statement is first parsed and the relational operations are represented as nodes in a query graph. The yellow nodes represent relational operations such as selection, aggregation (group by), union, insert, update, delete, etc.. The red nodes are leaf nodes representing data sources such as tables or table functions. The edges represent the flow of rows. Rows can flow in both directions. A downward flow represents a correlated reference in a lower sub-select, such as the correlated NOT EXISTS subquery in this example. Correlation requires that the lower sub-select be re-evaluated for each row provided by the downward edge.

A SELECT node can have multiple input edges which can either represent joins or subquery predicates. SELECT nodes also include SELECT list items including expressions and WHERE clause predicates.

Step 2: Query Rewrite

- Correlated NOT EXISTS subquery is converted to an anti-join
- Constant expressions are pre-computed
- Aggregation operations are unified

SELECT 1 → $C0 AS "TOTAL SHIPPING COST", ($C0/$C1) AS "AVG SHIPPING COST"

GROUP BY → SUM(CS_EXT_SHIP_COST) AS $C0, COUNT_BIG(CS_EXT_SHIP_COST) AS $C1

ANTIJOIN → SELECT Q5.CS_EXT_SHIP_COST CATALOG_RETURNS Q1 ANTIJOIN (SELECT 2) Q5 ON Q5.CS_ORDER_NUMBER = Q1.CR_ORDER_NUMBER)

D_DATE >= '04/01/2018' AND D_DATE <= '05/31/2018' AND CS_SHIP_DATE_SK = D_DATE_SK AND CS_SHIP_ADDR_SK = CA_ADDRESS_SK AND CA_STATE = 'NY' → **SELECT 2**

CATALOG RETURNS

DATE_DIM **CATALOG SALES** **CUSTOMER ADDRESS**

3 important query rewrites have occurred:

1) The correlated NOT EXISTS subquery has been rewritten as an anti-join. An anti-join is a type of join where only the rows that don't match are returned. The Db2 query runtime engine supports a efficient native anti-join.

2) The date expression "CAST('2018-4-01' AS DATE) + 60 DAYS" has been pre-computed as '05/31/2018'. This allows the optimizer to compute a more accurate selectivity estimate in a later phase.

3) The AVG aggregation function can be replaced with SUM/COUNT, re-using the existing SUM result

Db2 Query Rewrite Technology (1|2)

- **Heuristic-based decisions**
  - Push predicates close to data access
  - Decorrelate whenever possible
  - Transform subqueries to joins
  - Merge view definitions
- **Extensible architecture**
  - Set of rewrite rules and rule engine
  - Each rewrite rule is self-contained
  - Can add new rules and disable existing ones easily

13

Db2 Query Rewrite Technology (2|2)

- Rule engine with local cost-based decisions
- Rule engine iteratively transforms query until the query graph reaches a steady-state
- ~140 rules
- This presentation shows only a few examples

14

Query Rewrite - Operation Merge

- Goal: give the optimizer maximum latitude in its decisions
- Techniques:
  - View merge
    - makes additional join orders possible
    - can eliminate redundant joins
  - Subquery-to-join transformation
    - removes restrictions on join method/order
    - improves efficiency
  - Redundant join elimination
    - satisfies multiple references to the same table with a single scan
  - Shared aggregation
    - reduces the number of aggregation operations

Query Rewrite - Predicate Translation

- GOAL: optimal predicates
  - Distribute NOT (De Morgan's law)
    ... `WHERE NOT(COL1 = 10 OR COL2 > 3)`
    - becomes
    ... `WHERE COL1 <> 10 AND COL2 <= 3`
  - Predicate transitive closure
    - given predicates:
    `T1.C1 = T2.C2, T2.C2 = T3.C3, T1.C1 > 5`
    - add these predicates...
    `T1.C1 = T3.C3 AND T2.C2 > 5 AND T3.C3 > 5`
  - IN-to-OR conversion for Index ORing
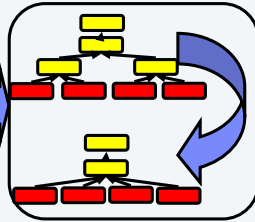 - and many more...

16

Query Optimization

- SQL compilation:
  - Query transformation (rewrite)
  - **Access plan generation**

Access section

Query transformation
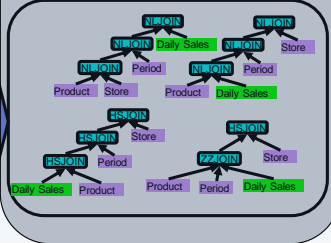
Access plan generation

```
SELECT ITEM_DESC, SUM(QUANTITY_SOLD),
    AVG(PRICE), AVG(COST)
  FROM PERIOD, DAILY_SALES, PRODUCT, STORE
WHERE
    PERIOD.PERKEY=DAILY_SALES.PERKEY AND
    PRODUCT.PRODKEY=DAILY_SALES.PRODKEY
     AND
    STORE.STOREKEY=DAILY_SALES.STOREKEY AND
     CALENDAR_DATE BETWEEN AND
      '01/01/2012' AND '04/28/2012' AND
     STORE_NUMBER='03' AND
     CATEGORY=72
GROUP BY ITEM_DESC
```

Thread 0
DSS
TQA (tq1)
AGG (complete)
BNO
EXT
Thread 1
TA (Product)
NLJN (Daily Sales)
NLJN (Period)
NLJN (Store)
AGG (partial)
TQB (tq1)
EXT
Thread 2
TA (DS_IX7)
EXT
Thread 3
TA (PER_IX2)
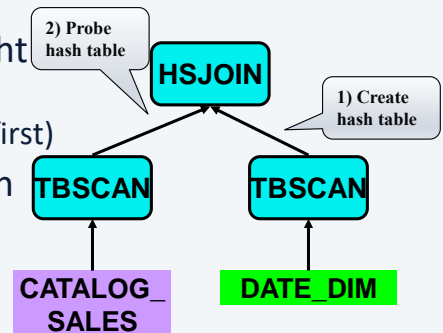EXT
Thread 4
TA (ST_IX1)
EXT

Dozens of query transformations

**Hundreds or thousands of access plan options**

17

Access Plan Generation

- An *Access Plan* represents a sequence of runtime operators used to execute the SQL statement
- Represented as a graph where each node is an operator and the edges represent the flow of data
- The order of execution is generally left to right
  - But there are some exceptions
  - (Hash join build table is on the RHS and is created first)
- Use the *explain facility* to see the access plan
  - (More on this later)

2) Probe hash table

HSJOIN

1) Create hash table

TBSCAN   **TBSCAN**

CATALOG_ SALES   **DATE_DIM**

18

Access Plan Generation

- Access plan generation occurs by scanning the Query Graph
- The access plan is built from the bottom up
  1. Build sub-plans for accessing tables first
     - Table scans, index scans
  2. Build plans for relational operations that consume those tables
     - Joins, GROUP BY, UNION, ORDER BY, DISTINCT
- Multiple preparatory Query Graph scans collect information to drive access plan generation
  - Interesting orders, DB partitioning and keys
  - Dependencies dictated by the Query Graph
    - i.e. correlation – must read table 1 before table 2

Access Plan Generation – Base Access and Joins
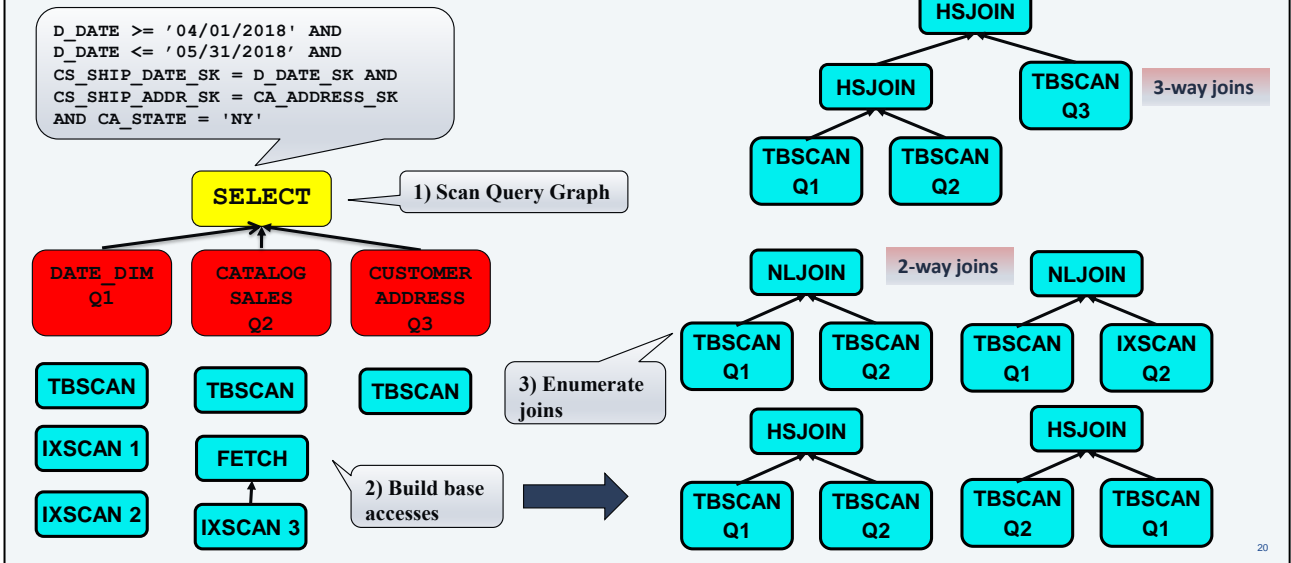
- For each relational operation in the query graph, evaluate runtime alternatives
  - Operation order
    - joins
    - predicate application – where?
    - aggregation – can be staged
  - Implementation to use:
    - table scan vs. index scan
    - nested-loop join vs. sort-merge join vs. hash join vs. zig-zag join

Access Plan Operators

- Access plan operators have *arguments* and *properties*
- Arguments tell Db2 runtime how they execute
  - e.g. sort key columns, partitioning columns, # of pages to prefetch, etc.
- Properties describe characteristics of the data stream
  - Columns projected
  - Order
  - Partitioning (DB partitioned environment)
  - Keys (uniqueness)
  - Predicates (filtering)
  - Maximum cardinality

21

Access Plan Operator Properties

- Properties can be exploited to improve performance
- Order, uniqueness and partitioning can be "valuable"
  - Because it takes work to create them
  - Order needs SORT ($$$)
  - Partitioning needs a table queue (TQ) ($$$)
  - Uniqueness needs a DISTINCT (or duplicate removing SORT) ($$$)
- More expensive sub-plans are retained if they possess an 'interesting' property
- Interestingness depends on the semantics of the query
  - Represented in the query graph

22

Access Plan Generation Considerations

- Where the access plan should execute:
  - Database partitioned systems
    - co-located, repartitioned or broadcast joins
  - Multi-core parallelism
    - degree of parallelism, parallelization strategies
  - Federated systems
    - push operations to remote servers
    - compensate in Db2
  - Column or row processing

23

Join Enumeration

- The search algorithm used to plan joins
- Search complexity depends on how tables are connected by predicates
- 2 methods:
  - Greedy
    - Most efficient, but not exhaustive
    - Could miss some good plans
  - Dynamic
    - Exhaustive, but expensive for large or highly connected join graphs

24

Dynamic Join Enumeration

{ CUSTOMER (Q1) }, { STORE_SALES (Q4) }
{ STORE (Q2) }, { STORE_SALES (Q4) }
{ DATE_DIM (Q3) }, { STORE_SALES (Q4) }

{ CUSTOMER (Q1) }, { DATE_DIM (Q3), STORE_SALES (Q4) }	**P4**
{ CUSTOMER (Q1) }, { STORE (Q2), STORE_SALES (Q4) }	P5
{ STORE (Q2) }, { DATE_DIM (Q3), STORE_SALES (Q4) }	**P6**
{ STORE (Q2) }, { CUSTOMER (Q1), STORE_SALES (Q4) }	P5
{ DATE_DIM (Q3) }, { STORE (Q2), STORE_SALES (Q4) }	**P6**
{ DATE_DIM (Q3) }, { CUSTOMER (Q1), STORE_SALES (Q4) }	**P4**

{ CUSTOMER (Q1) }, { STORE (Q2), DATE_DIM (Q3), STORE_SALES (Q4) }
{ STORE (Q2) }, { CUSTOMER (Q1), DATE_DIM (Q3), STORE_SALES (Q4) }
{ DATE_DIM (Q3) }, { CUSTOMER (Q1), STORE (Q2), STORE_SALES (Q4) }

CUSTOMER — STORE_SALES — STORE — DATE_DIM

25
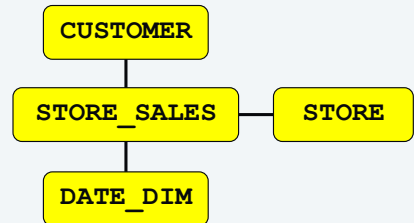
Greedy Join Enumeration

Only the cheapest join partition from each stage moves to the next stage
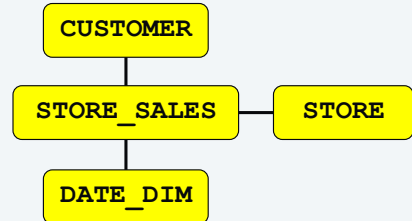
{ STORE_SALES (Q4) }, { CUSTOMER (Q1) }
{ STORE_SALES (Q4) }, { STORE (Q2) }
{ STORE_SALES (Q4) }, { DATE_DIM (Q3) }

{ CUSTOMER (Q1) }, { STORE (Q2), STORE_SALES (Q4) }
{ DATE_DIM (Q3) }, { STORE (Q2), STORE_SALES (Q4) }

{ CUSTOMER (Q1) }, { STORE (Q2), DATE_DIM (Q3), STORE_SALES (Q4) }

Optimization Classes and Join Enumeration

- Use optimization classes to control join enumeration method
- <u>Recommendation – use the default (5)</u>
- **Greedy join enumeration**
  - 0 - minimal optimization for OLTP
  - 1 - low optimization, no HSJOIN, IXSCAN, limited query rewrites
  - 2 - full optimization, limit space/time
    - use same query transforms & join strategies as class 5

- **Dynamic join enumeration**
  - 3 - moderate optimization, more limited plan space
  - 5 - self-adjusting full optimization (default)
    - uses all techniques with heuristics
  - 7 - full optimization
    - similar to 5, without heuristics
  - 9 - maximal optimization
    - spare no effort/expense
    - considers all possible join orders, including Cartesian products!

- Optimization requires processing time and memory
  - You can control resources applied to query optimization
  - (similar to the -O flag in a C compiler)
  - Special register, for dynamic SQL
    - SET CURRENT QUERY OPTIMIZATION = 1
  - Bind option, for static SQL
    - BIND YOURAPP.BND QUERYOPT 1
  - Database configuration parameter, for default
    - UPDATE DB CFG FOR <DB> USING DFT_QUERYOPT <N>
  - Static & dynamic SQL may use different values

Optimizer Cost Model

- Detailed model for each access plan operator
- Estimates the # of rows processed by each operator (*cardinality*)
  - Estimates predicate filtering (*filter factor* or *selectivity*)
  - Most important factor in determining an operator's cost
- Combine estimated runtime components to compute "cost":
  - CPU (# of instructions) +
  - I/O (random and sequential) +
  - Communications (# of IP frames, in parallel or Federated environments)

Simplified Costing Example (1|2)

- The cost model uses information from:
  - DBM config
  - System catalogs (SYSCAT.STOGROUPS, SYSCAT.TABLESPACES)
  - Catalog statistics (SYSSTAT.*)

Cost Model

Statistics

I/O cost =
(50000 * PAGESIZE / DEVICEREADRATE (MB/s))
 + (50000 / EXTENTSIZE * OVERHEAD (ms))

CPU cost = CPUSPEED * (
(#TBSCAN instructions * 50000) + (#predicate
instructions* 1000000) + (#data copy * 0.65 * 1000000))

Total Cost = I/O cost + CPU cost

TBSCAN
WHERE STATE = 'NC'

Customer

SELECTIVITY: 0.65
(frequent value statistics)

CARDINALITY: 1000000
FPAGES: 50000
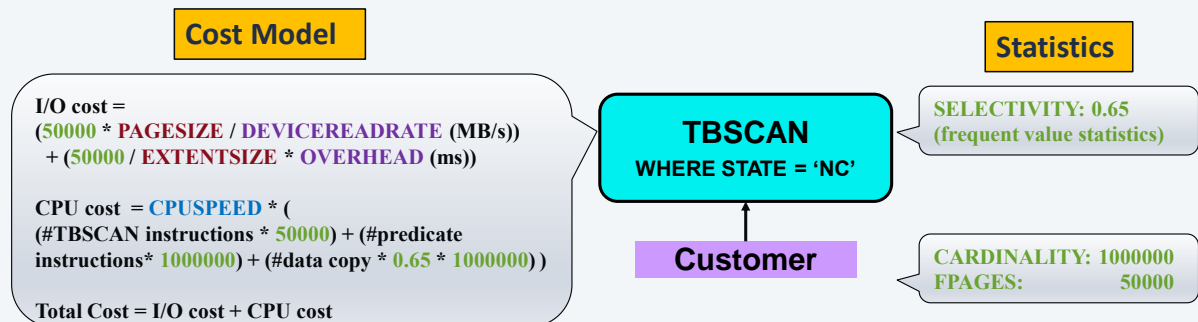
29

OVERHEAD

This attribute specifies the I/O controller time and the disk seek and latency time in milliseconds.

DEVICE READ RATE

This attribute specifies the device specification for the read transfer rate in megabytes per second. This value is used to determine the cost of I/O during query optimization. If this value is not the same for all storage paths, the number should be the average for all storage paths that belong to the storage group.

Simplified Costing Example (2|2)

- Each runtime cost component is modelled using milliseconds
- Runtime cost components are summed
- This does NOT represent elapsed time
  - Cost components typically execute concurrently
  - CPU and I/O parallelism
- Therefore total cost is in units of '**timeron**'
  - Just a made up name so it isn't mistaken for elapsed time

Optimizer Cost Model - Timerons

- Why is 'timeron' a better cost metric than elapsed time?
  - Timeron represents total system resource consumption
  - Preferred system metric assuming concurrent query / multi-user environment
  - Usually correlates to elapsed time too
- Some exceptions:
  - Approximate elapsed time is used for DB partitioned (MPP) systems
    - Total cost is average resource consumption per DB partition
    - Encourages access plans that execute on multiple DB partitions
  - Cost to get the first N rows
    - Used for OPTIMIZE FOR N ROWS/FETCH FIRST N ROWS ONLY or when 'piped' plans are desired

Costing for Database Partitioned Systems

- Cost is per DB partition
- Cost diminishes with more nodes -> encourages query parallelism
- Assumes a particular operator must process the same number of rows, globally

Execute on 4 nodes	Execute on 1 node
Cost = C * 250	Cost = C * 1000

TBSCAN	TBSCAN	TBSCAN	TBSCAN	TBSCAN
250 rows	250 rows	250 rows	250 rows	1000 rows

32

Optimizer Environment Awareness

- Speed of CPU
  - Determined automatically at instance creation time
  - Runs a timing program
  - Can be set manually (CPUSPEED DBM configuration parameter)
- Storage device characteristics
  - Used to model random and sequential I/O costs
  - I/O speed is based on :
    - I/O subsystem latency
    - Time to transfer data
  - Parameters are represented at the <u>storage group</u> and <u>table space</u> level
  - They are not set automatically by the DB2 server

Storage I/O Characteristics

- Storage groups
  - Latency: **OVERHEAD** (ms)
  - Data transfer speed: **DEVICE READ RATE** (MB/s)
- Table spaces:
  - Latency: **OVERHEAD** (ms)
  - Data transfer speed: **TRANSFERRATE** (ms/page)
    - Depends on the page size
- Default values for automatic storage table spaces are inherited from their underlying storage group
  - This is the recommended approach
  - Otherwise, be careful to adjust for different page sizes!

https://www.ibm.com/docs/en/db2/11.5?topic=design-table-space-impact-query-optimization

OVERHEAD number-of-milliseconds Specifies the I/O controller usage and disk seek and latency time. This value is used to determine the cost of I/O during query optimization. The value of number-of-milliseconds is any numeric literal (integer, decimal, or floating point). If this value is not the same for all storage paths, set the value to a numeric literal which represents the average for all storage paths that belong to the storage group.If the OVERHEAD clause is not specified, the OVERHEAD will be set to 6.725 milliseconds.

DEVICE READ RATE number-megabytes-per-second Specifies the device specification for the read transfer rate in megabytes per second. This value is used to determine the cost of I/O during query optimization. The value of number-megabytes-per-second is any numeric literal (integer, decimal, or floating point). If this value is not the same for all storage paths, set the value to a numeric literal which represents the average for all storage paths that belong to the storage group. If the DEVICE READ RATE clause is not specified, the DEVICE READ RATE will be set to the built-in default of 100 megabytes per second.

Optimizer Environment Awareness

- Buffer pool size
- Sort heap size
  - Used by sorts, hash join, index ANDing, hash aggregation and distincting
  - Main memory pool used by column-organized processing
- Communications bandwidth
  - To factor communication cost into overall cost, in DB partitioned environments
- Remote data source characteristics in a Federated environment
- Concurrency isolation level / locking
- Number of available locks

35

Planning and Modelling Predicate Application

- In general, optimizer tries to apply predicates as early as possible
  - Filter rows from stream to avoid unnecessary work
- However, some types of predicates can only be applied in certain locations during query execution
- There is a hierarchy of predicate application
- The explain facility shows where predicates are applied

Hierarchy of Predicate Application

Residual Predicates

Runtime engine

Salary > ALL
(SELECT...
FROM...
WHERE...)

Search Arguments (SARGs)

Name LIKE 'Lo%'

Buffer pages

Data Manager

Index sargable predicates

Start/stop keys

Index: (SSN,ID,TXID)

i-sarg: TXID = 9965
(applied to all qualifying keys)

Index Manager

Start/stop keys: SSN = '012-34-5678'

37

Cardinality Estimation

- *Cardinality* = number of rows
- The optimizer estimates the number of rows processed by each access plan operator
- Based on the number of rows in the table and the *filter factors* of applied predicates.
- This is the biggest impact on estimated cost!
- Catalog statistics are used to estimate filter factors and cardinality

Catalog Statistics

- Statistics are essential for query optimization
  - Used to compute access plan cost and cardinality
- Physical characteristic statistics
  - E.g. Number of pages in table, number of levels in an index
- Data attribute statistics
  - E.g. Number of rows in table, number of distinct values in a column, frequent values, quantiles
- Statistics collection methods:
  - RUNSTATS command
  - Automatically by Db2
    - Enabled using AUTO_RUNSTATS, AUTO_STMT_STATS DB config parameters
- Statistics are stored in the system catalogs
  - Visible in SYSSTAT and SYSCAT views:
    - TABLES, COLUMNS, INDEXES, COLDIST, COLGROUPS

When the SQL compiler optimizes SQL query plans, its decisions are heavily influenced by statistical information about the size of the database tables and indexes. The optimizer also uses information about the distribution of data in specific columns of tables and indexes if these columns are used to select rows or join tables. The optimizer uses this information to estimate the costs of alternative access plans for each query. Db2 will automatically collect statistics depending on how data changes and the statistical needs of queries. Statistics can also be collected manually using the RUNSTATS command. Statistical information is collected for specific tables, indexes and nicknames. The collected statistics are stored in the system catalog tables and can be queried using the SYSSTAT or SYSCAT catalog views.

Catalog Statistics Used by the Optimizer (1|3)

SYSSTAT.TABLES

Name	Description
CARD	Total number of rows in the table
NPAGES	Total number of pages on which the rows of the table exist
FPAGES	Total number of pages
MPAGES	Total number of pages for table metadata. (Columnar only)
OVERFLOW	Total number of overflow records in the table
ACTIVE_BLOCKS	Total number of active blocks in the table (MDC or ITC tables)
AVGROWSIZE	Average length (in bytes) of both compressed and uncompressed rows
AVGCOMPRESSEDROWSIZE	Average length (in bytes) of compressed rows in this table
AVGROWCOMPRESSIONRATIO	Average compression ratio for compressed rows in the table
PCTROWSCOMPRESSED	Compressed rows as a percentage of total number of rows in the table

Catalog Statistics Used by the Optimizer (2|3)

SYSSTAT.COLUMNS

Name	Description
COLCARD	Number of distinct values in the column
HIGH2KEY	Second-highest data value
LOW2KEY	Second-lowest data value
AVGCOLLEN	Avg. length in bytes when stored in DB memory or a temporary table
NUMNULLS	Number of null values in the column
SUB_COUNT	Avg. number of sub-elements in the column (LIKE predicate statistic)
SUB_DELIM_LENGTH	Avg. length of delimiters that separate each sub-element (LIKE predicate statistic)
AVGCOLLENCHAR	Avg. number of characters based on column collation
PCTENCODED	%age encoded values (column-organized table only)
AVGENCODEDCOLLEN	Avg. length when stored in DB memory (column-organized table only)

Catalog Statistics Used by the Optimizer (3|3)

SYSSTAT.INDEXES (not all statistics listed)

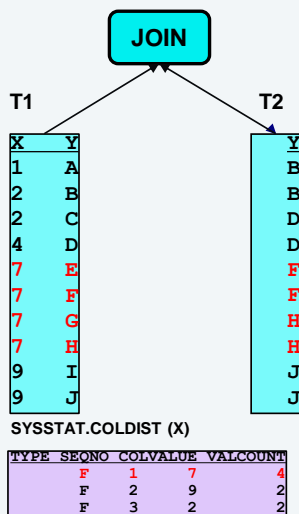| Name	Description
NLEAF	Number of leaf pages
NLEVELS	Number of index levels
FIRSTKEYCARD	Number of distinct first-key values
FIRSTnKEYCARD	Number of distinct keys using the first 2-4 columns of the index
FULLKEYCARD	Number of distinct values for the full index key
CLUSTERRATIO	Degree of data clustering with the index (non-detailed index statistics)
CLUSTERFACTOR	Finer measurement of the degree of clustering (detailed index statistics)
SEQUENTIAL_PAGES	Number of on-disk leaf pages in index key order with no gaps
DENSITY	Ratio of SEQUENTIAL_PAGES to number of prefetched pages (%age)
PAGE_FETCH_PAIRS	Data page fetches required for a range of buffer pool sizes

https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0001072.html

PAGE_FETCH_PAIRS -

A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.

Cardinality Estimation – Local and join predicates

SELECT * FROM T1, T2 WHERE T1.x = 7 AND T1.y = T2.y

JOIN

T1

T2

X	Y
1	A
2	B
2	C
4	D
7	E
7	F
7	G
7	H
9	I
9	J

Y
B
B
D
D
F
F
H
H
J
J

SYSSTAT.COLDIST (X)

TYPE	SEQNO	COLVALUE	VALCOUNT
F	1	7	4
F	2	9	2
F	3	2	2

Selectivity (T1.x = 7):   = 4/10
Using frequent value statistics

Selectivity (T1.y = T2.y):
= 1 / max(colcard(T1.y), colcard(T2.y))
= 1 / max(10,5)
= 1/10

Join predicate selectivity assumes:
Inclusion:
    All values in T2.y are included in domain of T1.y
Uniformity:
    Values are uniformly distributed in both columns

Result cardinality:
= Card(T1) * Card(T2) * sel(T1.x=7) * sel(T1.y=T2.y)
= 10 * 10 * 0.4 * 0.1
= 4
Actual: 4

Cardinality of T1: $C(T1) = 10$
Cardinality of T2: $C(T2) = 10$
Column cardinality of T1.Y: $CC(T1.Y) = 10$
Column cardinality of T2.Y: $CC(T2.Y) = 5$

Assuming even data distribution, there are the same number of duplicate values for each distinct value.
  $DC1 = C(T1)/CC(T1.Y) = 1$
  $DC2 = C(T2)/CC(T2.Y) = 2$

The column cardinality of the join result is min(CC(T1.Y),CC(T2.Y)). i.e. the number of distinct values that can occur in T1.Y and T2.Y after the join predicate is applied.
The number of rows returned by the join is the minimal join column cardinality times the number of duplicate values that can occur for each distinct value for each join column.

```
 min(CC(T1.Y),CC(T2.Y)) * DC1 * DC2 =

  min(CC(T1.Y),CC(T2.Y)) * (C(T1)/CC(T1.Y)) * (C(T2)/CC(T2.Y)) =

  min(CC(T1.Y) ,CC(T2.Y))
 --------------------------- * C(T1) * C(T2) =
  CC(T1.Y) * CC(T2.Y)


      1
 --------------------------  * C(T1) * C(T2)
 max(CC(T1.Y),CC(T2.Y))

 C(T1) * C(T2) = cardinality of Cartesian product of T1 and T2
     1
 ----------------------------- = selectivity of join predicate
 max(CC(T1.Y),CC(T2.Y))
```

The Explain Facility

- Internal phase of the optimizer that captures critical information used in selecting the query access plan
- Access plan information is written to a set of tables
- External tools to format explain table contents:
  - Db2 Data Management Console Visual Explain
    - GUI to render and navigate query access plans
    - Supersedes Data Server Manager Visual Explain
  - db2exfmt
    - Text-based output from the explain tables
   - Command-line interface

> They show the same information

The explain facility is used to display the query access plan chosen by the query optimizer to run an SQL statement. It contains extensive details about the relational operations used to run the SQL statement such as the plan operators, their arguments, order of execution, and costs. Since the query access plan is one of the most critical factors in query performance, it is important to be able to understand the explain facility output in order to diagnose query performance problems.

Explain information is typically used to:

- understand why application performance has changed
- evaluate performance tuning efforts

Db2 Data Management Console Visual Explain
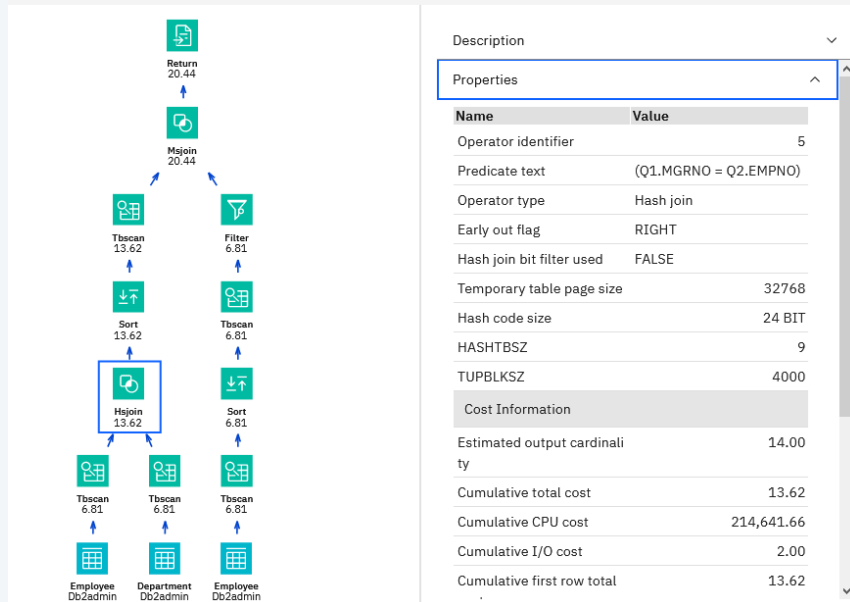
db2exfmt

Cardinality (rows)
Operator name
(Operator ID)
Cost (timerons)
I/O (pages)

```
                                        Rows
                                       RETURN
                                       (   1)
                                        Cost
                                         I/O
                                          |
                                       3.87404
                                       NLJOIN
                                       (  13)
                                       125.206
                                          5
                                 /-------+------\
                           0.968511              4
                            IXSCAN              FETCH
                            (  14)              (  15)
                            75.0966             100.118
                               3                   4
                               |               /----+---\
   Base table cardinality  4.99966e+06        4        1.99987e+07
                           INDEX: TPCD     IXSCAN     TABLE: TPCD
                           UXP_NMPK        (  16)     PARTSUPP
                                           75.1018
                                              3
                                              |
                                         1.99987e+07
                                    INDEX: TPCD.UXPS_PK2KSC
```

46

Explain Facility – Query Graph

• The Query Graph produced by query rewrite can be seen in the explain output as the *optimized SQL*

```
SELECT
  Q8.$C0 AS "total shipping cost", (Q8.$C0 / Q8.$C1) AS "total shipping cost"
FROM
  (SELECT SUM(Q7.CS_EXT_SHIP_COST), COUNT_BIG(Q7.CS_EXT_SHIP_COST)
   FROM
     (SELECT Q6.CS_EXT_SHIP_COST
      FROM
        (SELECT Q5.CS_EXT_SHIP_COST
         FROM TPCDS.CATALOG_RETURNS AS Q1
           RIGHT OUTER JOIN
           (SELECT Q4.CS_EXT_SHIP_COST, Q4.CS_ORDER_NUMBER
            FROM TPCDS.DATE_DIM AS Q2, TPCDS.CUSTOMER_ADDRESS AS Q3, TPCDS.CATALOG_SALES AS Q4
            WHERE
              ('04/01/2001' <= Q2.D_DATE) AND (Q2.D_DATE <= '05/31/2001') AND
              (Q4.CS_SHIP_DATE_SK = Q2.D_DATE_SK) AND (Q4.CS_SHIP_ADDR_SK = Q3.CA_ADDRESS_SK) AND
              (Q3.CA_STATE = 'NY')
           ) AS Q5
           ON (Q5.CS_ORDER_NUMBER = Q1.CR_ORDER_NUMBER)
        ) AS Q6
     ) AS Q7
  ) AS Q8
```

John Hornibrook
IBM Canada
jhornibr@ca.ibm.com

John is a Senior Technical Staff Member responsible for relational database query optimization on IBM's distributed platforms. This technology is part of Db2 for Linux, UNIX and Windows, Db2 Warehouse, Db2 on Cloud, IBM Integrated Analytics System (IIAS) and Db2 Big SQL. John also works closely with users to help them fully realize the benefits of IBM's relational DB technology products.