IBM System z Technology Summit

DB2 10 for z/OS Query Technology features

Andrei Lurie

alurie@us.ibm.com

Disclaimer

Agenda

- **DB2 10 Optimizer**
  - Plan management
  - Hints/Bind options
  - Explain
  - Dynamic Statement Caching
  - Optimizer costing
  - Runtime query performance
- **DB2 10 SQL/Application Enablement**
  - SQL table functions and non-inline SQL scalar functions
  - Implicit casting
  - Datetime constants and TIMESTAMP WITH TIMEZONE
  - Extended indicator variables

Plan Management Notes (DB2 9)

- **REBIND PACKAGE options:**
  - PLANMGMT (BASIC)

  2 copies: Current and Previous
  - PLANMGMT (EXTENDED)

  3 copies: Current, Previous, Original

- **REBIND PACKAGE options:**
  - SWITCH(PREVIOUS)

  Switch between current & previous
  - SWITCH(ORIGINAL)

  Switch between current & original

- **Most bind options can be changed at REBIND**
  - *But a few must be the same …*

- **FREE PACKAGE …**
  - PLANMGMTSCOPE(ALL) – Free package completely
  - PLANMGMTSCOPE(INACTIVE) – Free old copies

- **Catalog support**
  - SYSPACKAGE reflects active copy **only**
  - SYSPACKDEP reflects dependencies of all copies
  - Other catalogs (SYSPKSYSTEM, …) reflect metadata for all copies

- **Invalidation and Auto Bind**
  - Each copy invalidated separately

DB2 10 Updates to Plan Management

- **SYSIBM.SYSPACKCOPY**
  - New catalog table
  - Hold SYSPACKAGE-style metadata for any previous or original package copies
  - No longer need to SWITCH to see information on inactive copies

- **APRETAINDUP option of REBIND**
  - Default YES
    - Retain duplicate for BASIC or EXTENDED
  - Optional NO
    - Do not retain duplicate access path as PREVIOUS or ORIGINAL
      - PREVIOUS/ORIGINAL must be from DB2 9 or later

- **Native SQL Stored Procedure packages are supported**

Retrieving Access Path with EXPLAIN(NO)

- **EXPLAIN PACKAGE**
  - Extract existing PLAN_TABLE information for packages
    - **NOT a new explain**
    - The package/copy must be created on DB2 9 or later
  - Useful if you didn’t BIND with EXPLAIN(YES)
    - Or PLAN_TABLE entries are lost

```
>>-EXPLAIN----PACKAGE----------->

>>-----COLLECTION--collection-name--PACKAGE--package-name--------->

>----+---------------------------+----+-------------------+-------->
     |                           |    |                   |
     +---VERSION-version-name---+    +---COPY--copy-id---+
```

- COPY-ID can be ‘CURRENT’, ‘PREVIOUS’, ‘ORIGINAL’

What-if? BIND

- BIND package to see what's new
- Bind package **EXPLAIN(ONLY)** and/or **SQLERROR(CHECK)**
  - Existing package copies are not overwritten
    - Performs explain or syntax/semantic error checks on SQL
  - Requires BIND, BINDAGENT, or EXPLAIN privilege.
  - Supported for BIND only
    - Not REBIND
    - Targeted to application changes
      - E.g. Development environment is DB2 LUW, production DB2 for z/OS

Access Path Stability with statement level hints

- **Current limitations in hint matching**
  - QUERYNO is used to link queries to their hints – a bit fragile
  - For dynamic SQL, require a change to apps – can be impractical
- **New mechanisms:**
  - Associate query text with its corresponding hint … more robust
  - Hints enforced for the entire DB2 subsystem
    - irrespective of static vs. dynamic, etc.
  - Hints integrated into the access path repository

- **PLAN_TABLE isn't going away**
- **Only the "hint lookup" mechanism is being improved.**

Statement level hints (cont.)

- **Steps to use new hints mechanism**
  - Populate a user table DSN_USERQUERY_TABLE with query text and QUERYNO
  - Populate PLAN_TABLE with the corresponding hints
  - Run new command BIND QUERY
    - To integrate the hint into the repository.
  - FREE QUERY command can be used to remove the hint.

Statement-level BIND options

- **Statement-level granularity may be required rather than:**
  - Subsystem level ZPARMs (STARJOIN, SJTABLES, MAX_PAR_DEGREE)
  - Package level BIND options (REOPT, DEF_CURR_DEGREE)
- **For example**
  - Only one statement in the package needs REOPT(ALWAYS)
- **New mechanism for statement-level bind options:**
  - Similar to mechanism used for hints
  - DSN_USERQUERY_TABLE can also hold per-statement options

Literal Replacement

- **Dynamic SQL with literals can now be re-used in the cache**
  - Literals replaced with &
    - Similar to parameter markers but not the same
- **To enable:**
  - Put CONCENTRATE STATEMENTS WITH LITERALS in the PREPARE ATTRIBUTES clause
  - Or set LITERALREPLACEMENT in the ODBC initialization file
  - Or set the keyword enableLiteralReplacement='YES' in the JCC Driver
- **Lookup Sequence**
  - Original SQL with literals is looked up in the cache
  - If not found, literals are replaced and new SQL is looked up in the cache
    - Additional match on literal usability
    - Can only match with SQL stored with same attribute, not parameter marker
  - If not found, new SQL is prepared and stored in the cache

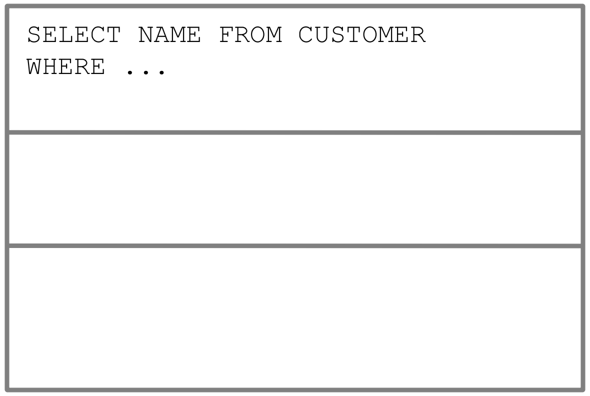
Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = 123456
```

SQL statement comes in …

Statement Cache (DSC)

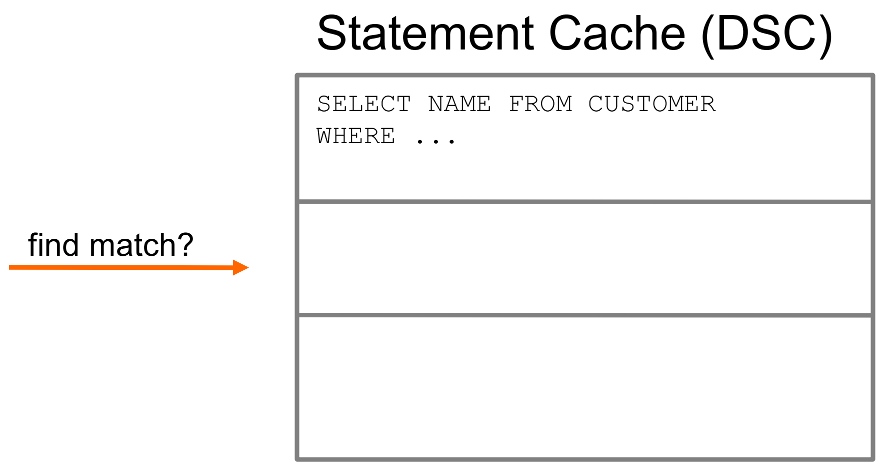


```
SELECT NAME FROM CUSTOMER
WHERE ...
```

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = 123456
```

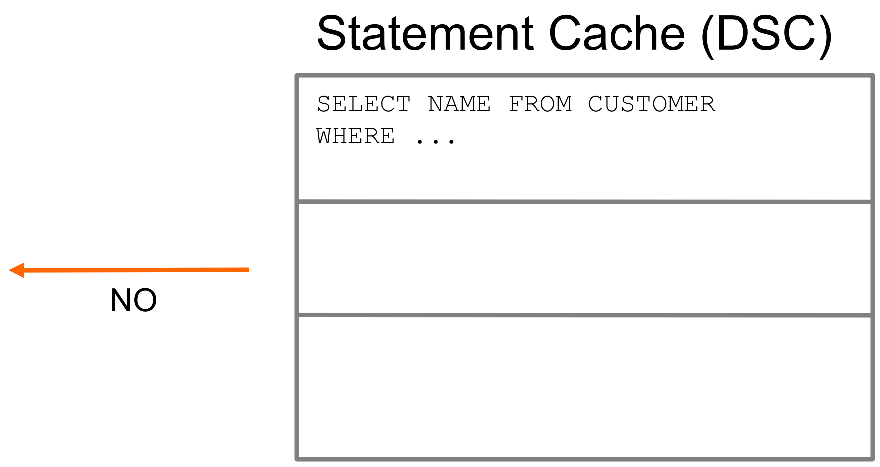


First lookup

Literal Replacement ...

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = 123456
```

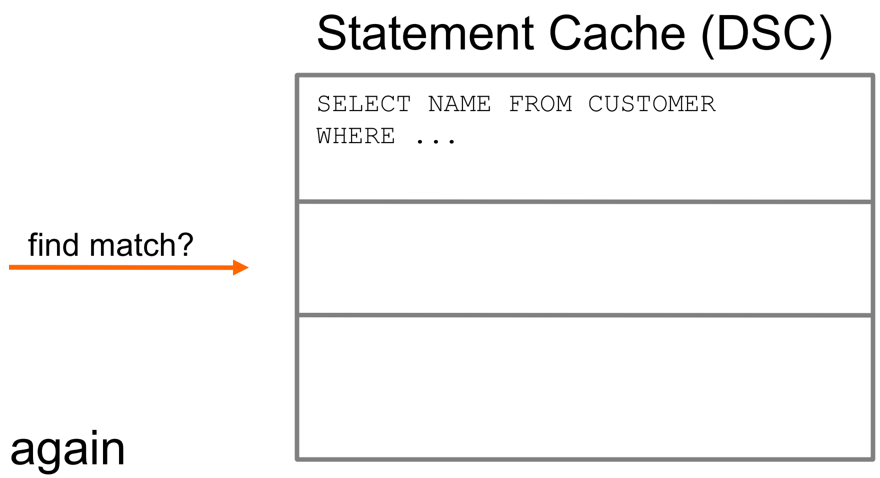


```
SELECT NAME FROM CUSTOMER
WHERE ...
```

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```



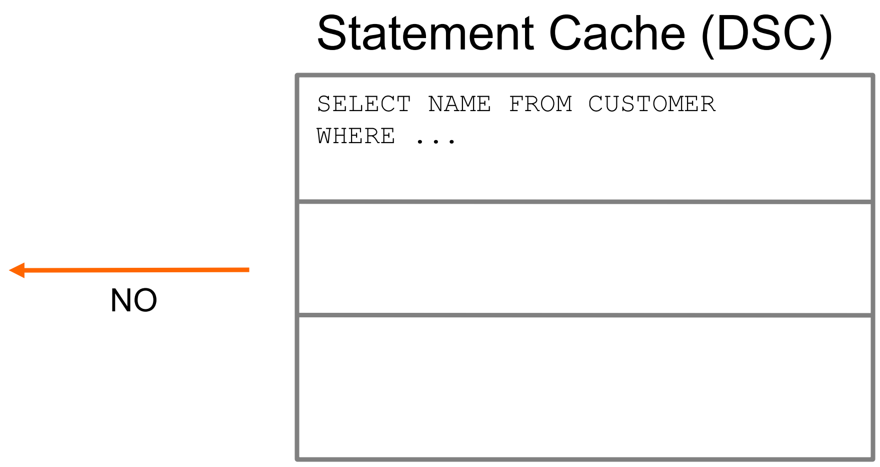
```
SELECT NAME FROM CUSTOMER
WHERE ...
```

- Replace literal with & and look up DSC again

Literal Replacement ...

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```



```
SELECT NAME FROM CUSTOMER
WHERE ...
```

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

Statement Cache (DSC)

```
SELECT NAME FROM CUSTOMER
WHERE ...
```

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

- PREPARE and save into DSC

(note that PREPARE is done on & “version”, not on original statement with literal)

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = 678900
```

- new statement (new literal)

Statement Cache (DSC)

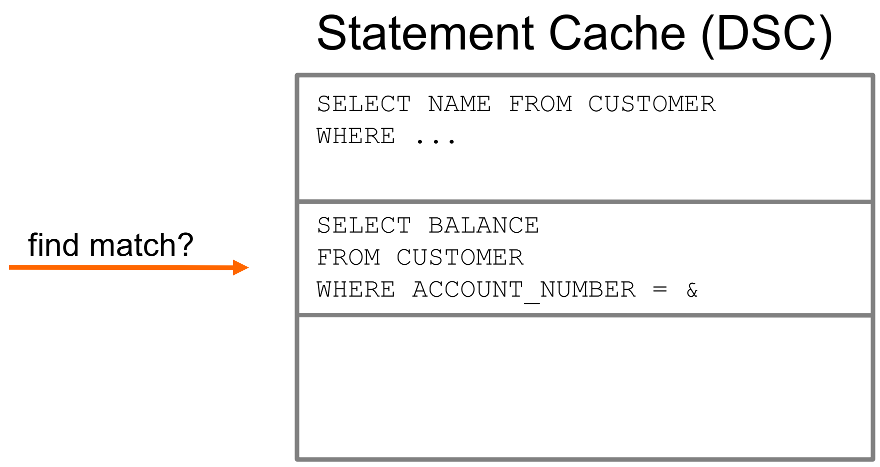
```
SELECT NAME FROM CUSTOMER
WHERE ...
```

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = 678900
```

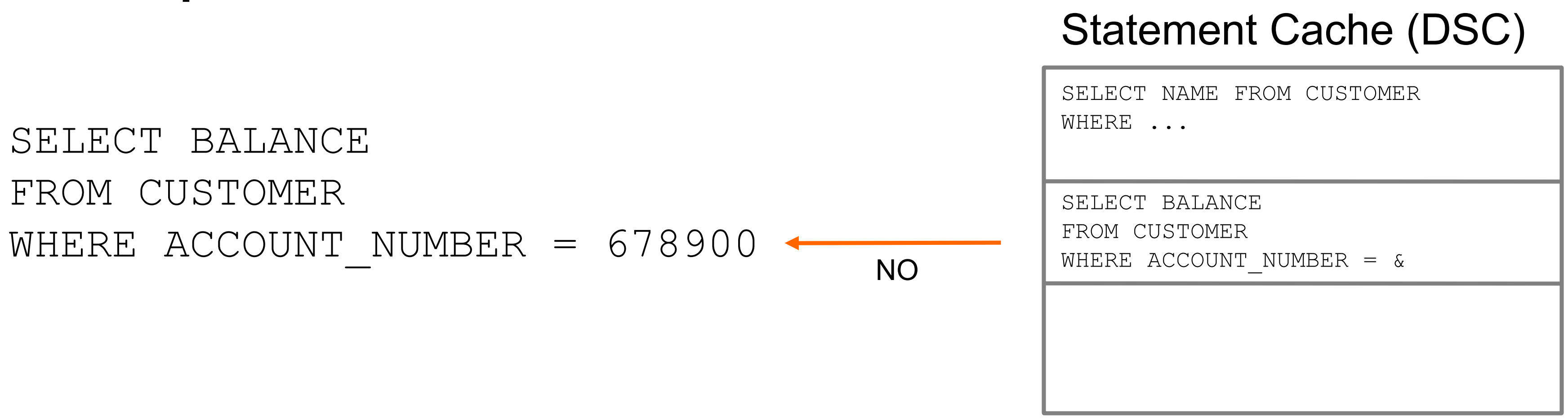


```
SELECT NAME FROM CUSTOMER
WHERE ...
```

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

Literal Replacement ...

- **Example:**



```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = 678900
```

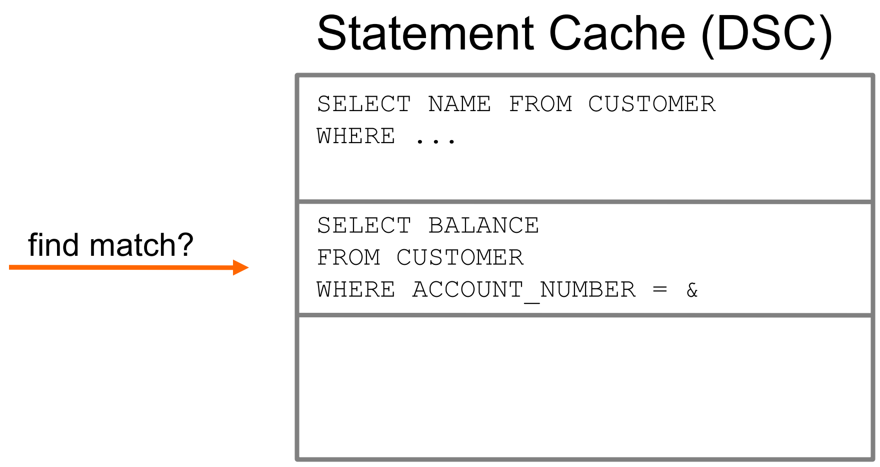
```
SELECT NAME FROM CUSTOMER
WHERE ...
```

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```



```
SELECT NAME FROM CUSTOMER
WHERE ...
```

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

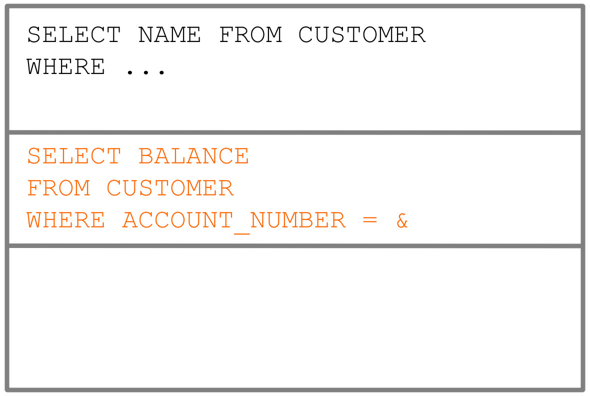
- Replace literal with & and look up DSC again

Literal Replacement …

- **Example:**

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

Statement Cache (DSC)



```
SELECT NAME FROM CUSTOMER
WHERE ...
```

```
SELECT BALANCE
FROM CUSTOMER
WHERE ACCOUNT_NUMBER = &
```

- Match found :) avoids PREPARE

Literal Replacement …

- **Performance Expectation**
  - Using parameter marker still provides best performance
  - Biggest performance gain for repeated SQL with different literals
  - NOTE: Access path is not optimized for literals
    - True for parameter markers/host variables today
    - Need to use REOPT for that purpose

Histogram Statistics (DB2 9)

- **RUNSTATS will produce equal-depth histogram**
  - Each quantile (range) will have approx same number of rows
    - Not same number of values
  - Another term is range frequency

- **Example**
    - 1, 3, 3, 4, 4, 6, 7, 8, 9, 10, 12, 15 (sequenced)
  - Lets cut that into 3 quantiles.
    - 1, 3, 3, 4 ,4 6,7,8,9 10,12,15

Seq No	Low Value	High Value	Cardinality	Frequency
1	**1**	**4**	**3**	**5/12**
2	**6**	**9**	**4**	**4/12**
3	**10**	**15**	**3**	**3/12**

Histogram Statistics Notes

- **RUNSTATS**
  - Maximum 100 quantiles for a column
  - Same value columns WILL be in the same quantile
  - Quantiles will be similar size but:
    - Will try to avoid big gaps inside quantiles
    - Highvalue and lowvalue may have separate quantiles
    - Null WILL have a separate quantile
- **Supports column groups as well as single columns**
- **Think “frequencies” for high cardinality columns**

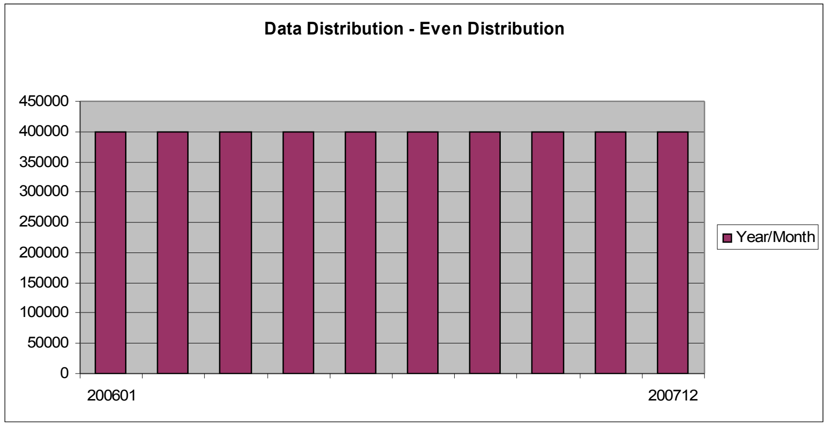
Histogram Statistics Example

- **Customer uses INTEGER (or VARCHAR) for YEAR-MONTH**

WHERE YEARMONTH BETWEEN 200601 AND 200612

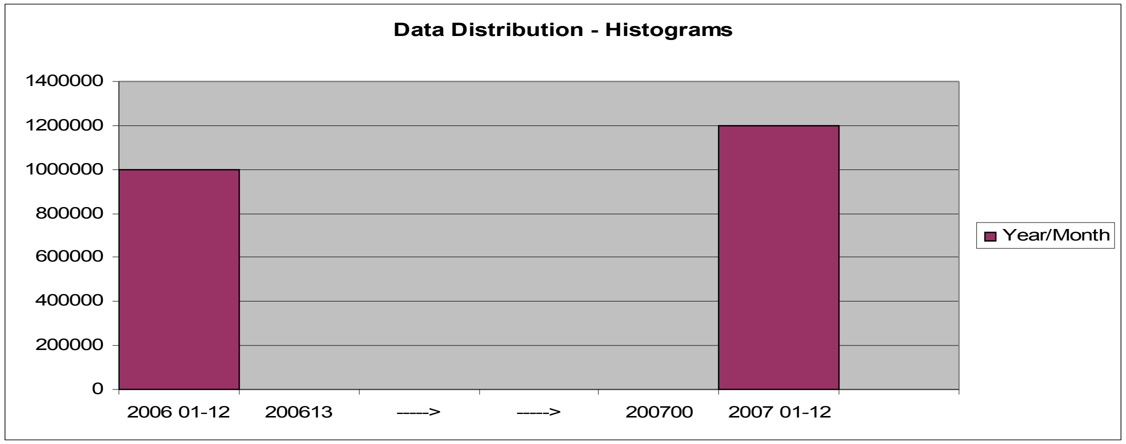
- Assuming data for 2006 & 2007
  - FF = (high-value – low-value) / (high2key – low2key)
  - FF = (200612 – 200601) / (200711 – 200602)
  - **10% of rows estimated to return**

Data assumed as evenly distributed between low and high range



Histogram Statistics Example

- Example (cont.)
  - Data only exists in ranges 200601-12 & 200701-12
    - Collect via histograms
      - 45% of rows estimated to return



No data between 200613 & 200700

WHERE YEARMONTH BETWEEN 200601 AND 200612

Autonomic Statistics Solution Overview

- **Autonomic Statistics is implemented through a set of Stored Procedures**
  - *Stored procedures are provided to enable administration tools and packaged applications to automate statistics collection.*
    - ADMIN_UTL_MONITOR
    - ADMIN_UTL_EXECUTE
    - ADMIN_UTL_MODIFY
  - Working together, these SP’s
    - Determine what stats to collect
    - Determine when stats need to be collected
    - Schedule and Perform the stats collection
    - Records activity for later review
  - *See Chapter 11 "Designing DB2 statistics for performance" in the DB2 10 for z/OS Performance Monitoring and Tuning Guide for details on how to configure autonomic monitoring directly within DB2.*

RUNSTATS Simplification/Performance Overview

- **RUNSTATS options to SET/UPDATE/USE a stats profile**
  - Predefine set of options (one per table)
    - RUNSTATS … TABLE tbl COLUMN(C1) … **SET PROFILE**
      - Alternatively use **SET PROFILE FROM EXISTING STATS**
    - RUNSTATS … TABLE tbl COLUMN(C5)… **UPDATE PROFILE**
    - RUNSTATS … TABLE tbl **USE PROFILE**

- **New option for page-level sampling**
  - But what percentage of sampling to use?
    - RUNSTATS … TABLE tbl **TABLESAMPLE SYSTEM AUTO**

DB2 10 - Minimizing Optimizer Challenges

- **Potential causes of sub-optimal plans**
  - Insufficient statistics
  - Unknown literal values used for host variables or parameter markers

- **DB2 10 Optimizer will evaluate the risk for each predicate**
  - For example: WHERE BIRTHDATE < ?
    - Could qualify 0-100% of data depending on literal value used
  - As part of access path selection
    - Compare access paths with close cost and choose lowest risk plan

Improvements to predicate application

- **Major enhancements to OR and IN predicates**
  - Improved performance for AND/OR combinations and long IN-lists
    - General performance improvement to stage 1 predicate processing
  - IN-list matching
    - Matching on multiple IN-lists
    - Transitive closure support for IN-list predicates
    - List prefetch support
  - SQL pagination
    - Single index matching for complex OR conditions

- **Many stage 2 expressions to be executed at stage 1**
  - Stage 2 expressions eligible for index screening
    - Not applicable for list prefetch
    - e.g. WHERE SUBSTR(SSN,8,4) = :hv, WHERE C1*2 > :hv ...
  - Externalized in DSN_FILTER_TABLE column PUSHDOWN

IN-list Table - Table Type 'I' and Access Type 'IN'

- The IN-list predicate will be represented as an in-memory table if:
  - **List prefetch is chosen, OR**
  - **More than one IN-list is chosen as matching.**

  - **The EXPLAIN output associated with the in-memory table will have:**
    - **New Table Type: TBTYPE – ‘I’**
    - **New Access Type: ACTYPE – ‘IN’**

```
SELECT *
   FROM T1
   WHERE T1.C1 IN (?, ?, ?);
```

QBNO	PLANNO	METHOD	TNAME	ACTYPE	MC	ACNAME	QBTYPE	TBTYPE	PREFETCH
1	1	0	DSNIN001(01)	*IN*	0		SELECT	*I*	
1	2	1	T1	I	1	T1_IX_C1	SELECT	T	*L*

IN-list Predicate Transitive Closure (PTC)

```
SELECT *
FROM T1, T2
WHERE T1.C1 = T2.C1
  AND T1.C1 IN (?, ?, ?)

  AND T2.C1 IN (?, ?, ?) ←   Optimizer can generate
                       this predicate via PTC
```

The idea:
If A = B and B = C
then A = C

- **Without IN-list PTC (DB2 9)**
  – Optimizer will be unlikely to consider T2 as the first table accessed

- **With IN-list PTC (DB2 10)**
  – Optimizer can choose to access T2 or T1 first.

SQL Pagination (range list index scan)

- **Targets 2 types of queries**
  - Cursor scrolling (pagination) SQL
    - Retrieve next n rows
      - Common in COBOL/CICS and any screen scrolling application
    - Not to be confused with “scrollable cursors”
  - Complex OR predicates against the same columns
    - Common in SAP

- **In both cases:**
  - The OR (disjunct) predicate refers to a single table only.
  - Each OR predicate can be mapped to the same index.
  - Each disjunct has at least one matching predicate.

Simple scrolling – Index matching and ORDER BY

- Scroll forward to obtain the next 20 rows
  - Assumes index is available on (LASTNAME, FIRSTNAME)
  - WHERE clause may appear as:

```
WHERE (LASTNAME='JONES' AND FIRSTNAME>'WENDY')
   OR (LASTNAME>'JONES')
ORDER BY LASTNAME, FIRSTNAME;
```

  - DB2 10 supports
    - Single matching index access with sort avoided
  - DB2 9 requires
    - Multi-index access, list prefetch and sort
    - OR, extra predicate (AND LASTNAME >= ‘JONES’) for matching single index access and sort avoidance

Complex OR predicates against same index

- Given WHERE clause
  - And index on one or both columns

```
WHERE (LASTNAME='JONES' AND FIRSTNAME='WENDY')
   OR (LASTNAME='SMITH' AND FIRSTNAME='JOHN');
```

- DB2 9 requires
  - Multi-index access with list prefetch

- DB2 10 supports
  - Matching single index access – no list prefetch
  - Or, Multi-index access with list prefetch

Minimizing impact of RID failure

- **RID overflow can occur for**
  - – Concurrent queries each consuming shared RID pool
  - – Single query requesting > 25% of table or hitting RID pool limit

- **DB2 9 will fallback to tablespace scan***

- **DB2 10 will continue by writing new RIDs to workfile**
  - – Work-file usage may increase
    - Mitigate by increasing RID pool size (default increased in DB2 10).
    - MAXTEMPS_RID zparm for maximum WF usage for each RID list

* Hybrid join can incrementally process. Dynamic Index ANDing will use WF for failover.

Removal Of Parallelism Restrictions

- Allow parallelism if a parallel group contains a work file
  - DB2 generates temporary work file when view or table expression is materialized
  - This type of work file can not be shared among child tasks in previous releases of DB2, hence parallelism is disabled
  - DB2 10 will make the work file shareable
    - only applies to CP mode parallelism and no full outer join case

- Support parallelism for multi-row fetch (READ ONLY cursors)
  - In previous releases parallelism is disabled for the last (top level query block) select
  - Example: SELECT * FROM CUSTOMER;

Parallelism Enhancements - Effectiveness

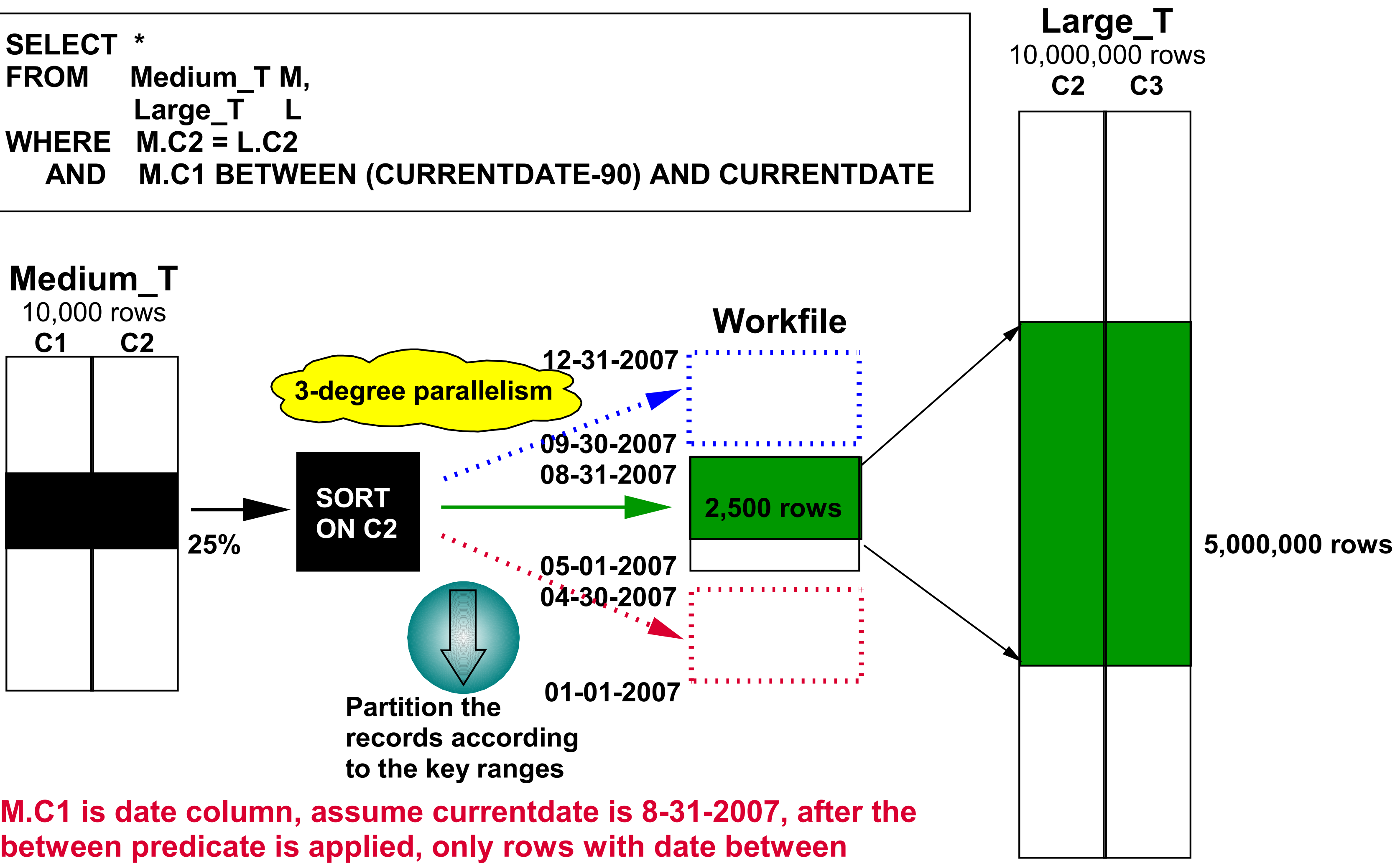
- **Previous Releases of DB2 may use Key Range Partitioning**
  - Key Ranges Decided at Bind Time
  - Based on Statistics (low2key, high2key, column cardinality)
    - Assumes uniform data distribution
    - Histograms can help
      - But rarely collected
  - If Statistics are outdated or data is not uniformly distributed what happens to performance?

Parallelism Enhancements - Effectiveness

- **Previous Releases of DB2 may use Key Range Partitioning**
  - Key Ranges Decided at Bind Time
  - Based on Statistics (low2key, high2key, column cardinality)
    - Assumes uniform data distribution
    - Histograms can help
      - But rarely collected
  - If Statistics are outdated or data is not uniformly distributed what happens to performance?

Key range partition – before DB2 10

```
SELECT  *
FROM     Medium_T M,
         Large_T    L
WHERE   M.C2 = L.C2
   AND    M.C1 BETWEEN (CURRENTDATE-90) AND CURRENTDATE
```



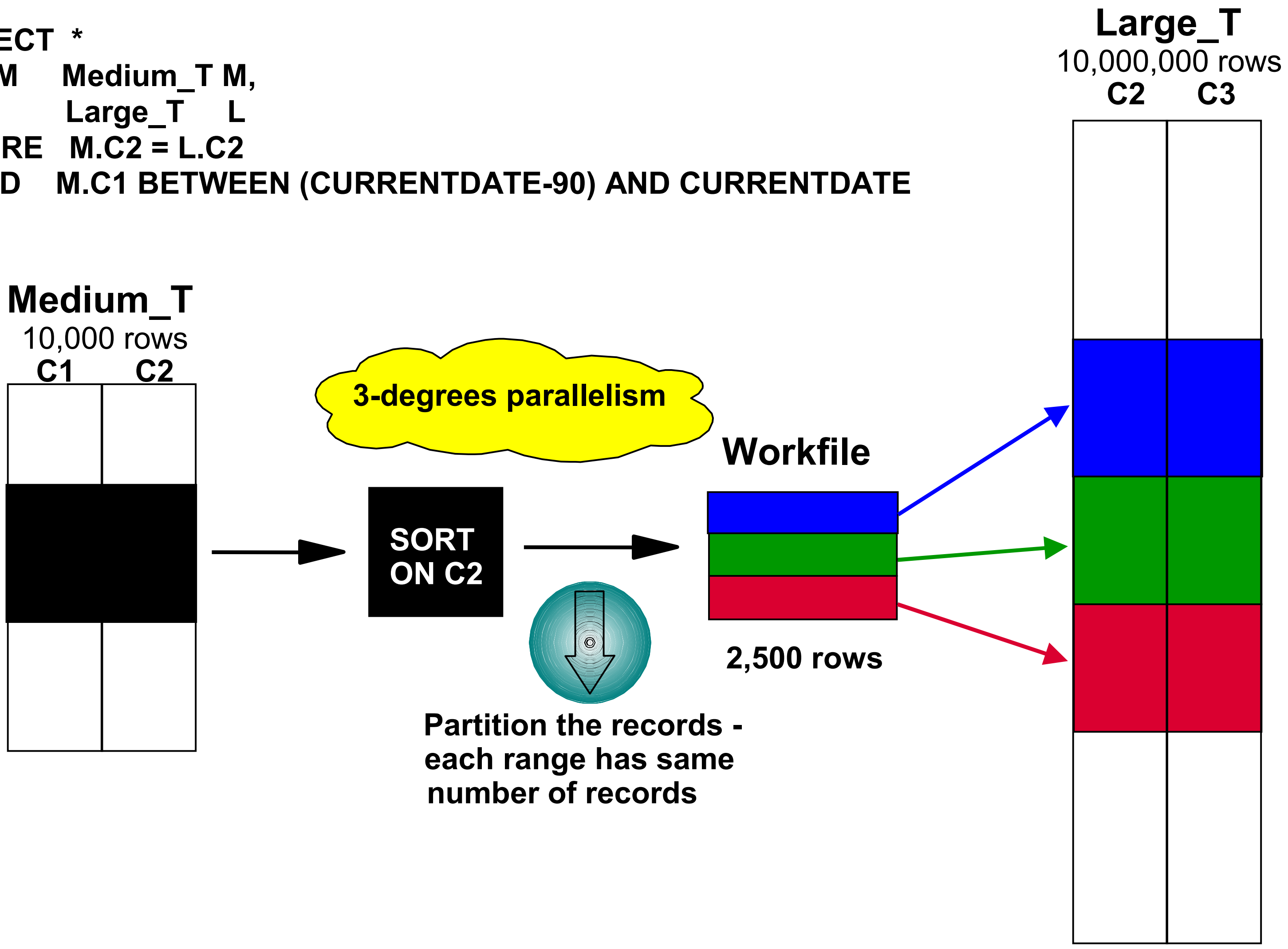
M.C1 is date column, assume currentdate is 8-31-2007, after the between predicate is applied, only rows with date between 06-03-2007 and 8-31-2007 survived, but optimizer chops up the key ranges within the whole year after the records are sorted :-(

Parallelism Effectiveness – Record range

- **DB2 10 can use** Dynamic record range partitioning
  - Results divided into ranges with equal number of records
  - Division doesn't have to be on the key boundary
    - Unless required for group by or distinct function
  - Record range partitioning is dynamic
    - no longer based on the key ranges decided at bind time
  - Now based on number of composite records and parallel degree
    - Data skew, out of date statistics etc. will not have any effect on performance

Dynamic record range partition

```
 SELECT  *
FROM     Medium_T M,
          Large_T    L
WHERE   M.C2 = L.C2
  AND    M.C1 BETWEEN (CURRENTDATE-90) AND CURRENTDATE
```



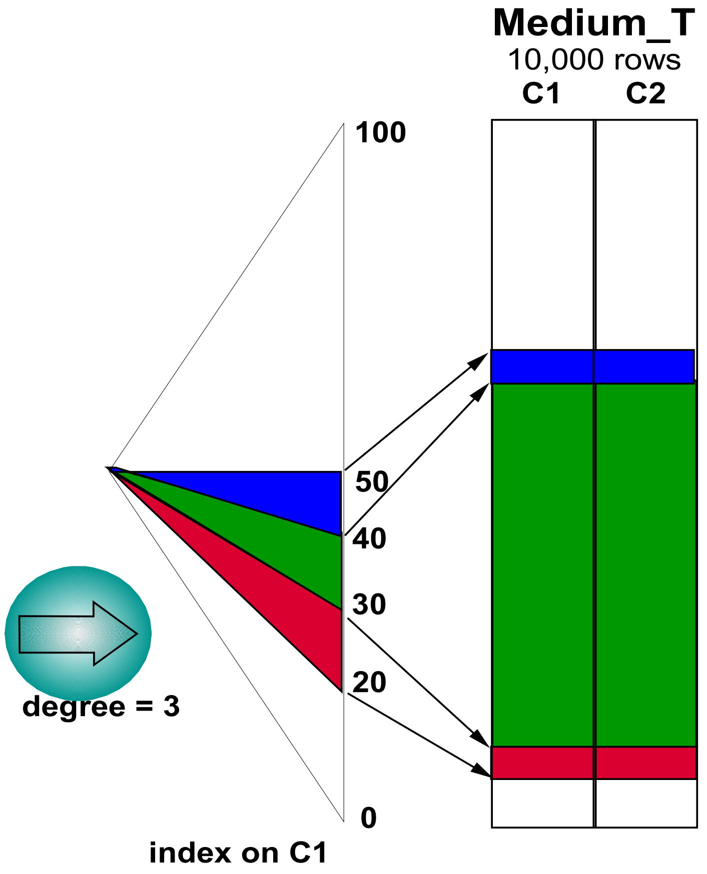
Parallelism Effectiveness - Straw Model

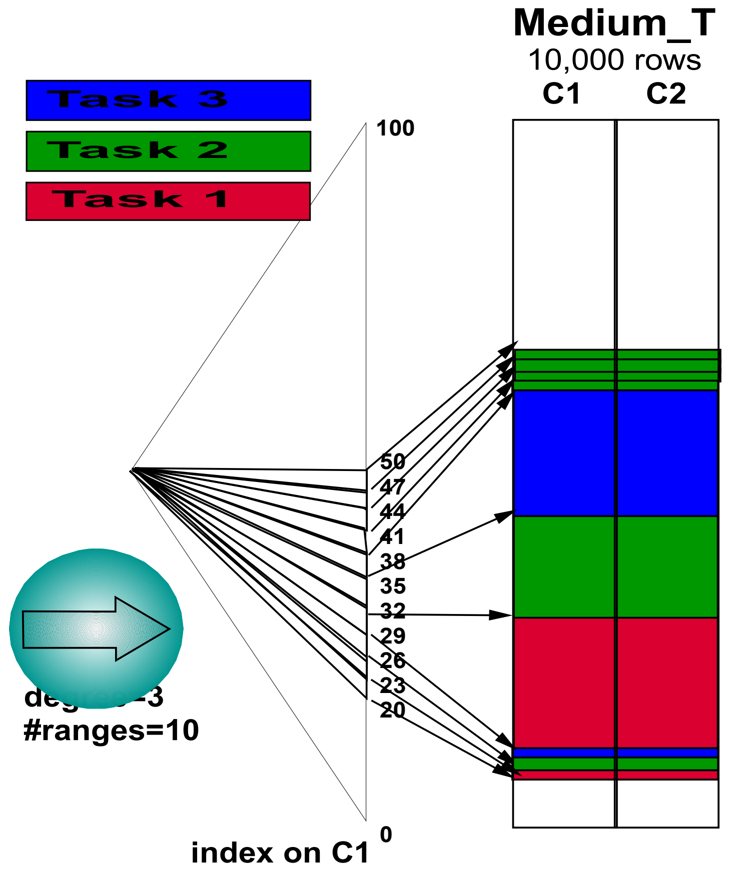
- **Previous releases of DB2 divide the number of keys or pages by the number representing the parallel degree**
  - One task is allocated per degree of parallelism
  - The range is processed and the task ends
  - Tasks may take different times to process

- **DB2 10 can use the Straw Model workload distribution method**
  - More key or page ranges will be allocated than the number of parallel degrees
  - The same number of tasks as before are allocated (same as degree)
  - Once a task finishes it’s smaller range it will process another range
  - Even if data is skewed this new process should make processing faster

STRAW Model

```
SELECT  *
FROM     Medium_T M
WHERE   M.C1 BETWEEN 20 AND 50
```





Divided in key ranges before DB2 10

Divided in key ranges with Straw Model

Agenda

- DB2 10 Optimizer
  - Plan management
  - Hints/Bind options
  - Explain
  - Dynamic Statement Caching
  - Optimizer costing
  - Runtime query performance
- **DB2 10 SQL/Application Enablement**
  - SQL table functions and SQL scalar functions
  - Implicit casting
  - Datetime constants / TIMESTAMP WITH TIME ZONE
  - Extended indicator variables

SQL table functions

- **DB2 10 supports simple SQL table functions**
  - Can use single RETURN control statement in function body
  - Can define parameter as: distinct type, transition table
  - No package is generated; reference is replaced similar to inline SQL scalar

```
CREATE FUNCTION TRYTABLE(P1 CHAR(3))
  RETURNS TABLE(FIRSTNAME VARCHAR(12), LASTNAME VARCHAR(15))
  RETURN SELECT FIRSTNME, LASTNAME FROM DSN8A10.EMP WHERE WORKDEPT = P1;


SELECT * FROM TABLE(TRYTABLE('A00')) X;

     +---------------------------------+
     |  FIRSTNAME    |     LASTNAME     |
     +---------------------------------+
     | CHRISTINE     | HAAS             |
     | VINCENZO      | LUCCHESI         |
     | SEAN          | O'CONNELL        |
     | DIAN          | HEMMINGER        |
     | GREG          | ORLANDO          |
     +---------------------------------+
```

SQL non-inline scalar functions

DB2 10 differentiates between **inline** SQL scalar and **non-inline** SQL scalar functions

- Inline: functions with the same capability as in prior releases
  - No package is generated; reference is replaced by the expression
- Non-inline
  - Package is created (note REBIND PACKAGE rebinds only SQL in function body; does not rebind SQL control statements – use ALTER FUNCTION REGENERATE).

    e.g.:
```
CREATE FUNCTION TEST_FN(P1 TABLE LIKE EMP AS LOCATOR)
 RETURNS INT   LANGUAGE SQL …
 BEGIN
  DECLARE VAR1 INT;
  SET VAR1 = (SELECT ID FROM TABLE(P1 LIKE EMP));
  IF VAR1 <> 42 THEN INSERT INTO REPORT VALUES(...);
  END IF;
  RETURN VAR1;
 END#
```

SQL non-inline scalar functions

DB2 10 differentiates between **inline** SQL scalar and **non-inline** SQL scalar functions

– Inline: functions with the same capability as in prior releases

  - No package is generated; reference is replaced by the expression

– Non-inline

  - Package is created (note REBIND PACKAGE rebinds only SQL in function body; does not rebind SQL control statements – use ALTER FUNCTION REGENERATE).

    e.g.:

```
CREATE FUNCTION TEST_FN(P1 TABLE LIKE EMP AS LOCATOR)
 RETURNS INT   LANGUAGE SQL …
 BEGIN
  DECLARE VAR1 INT;
  SET VAR1 = (SELECT ID FROM TABLE(P1 LIKE EMP));
  IF VAR1 <> 42 THEN INSERT INTO REPORT VALUES(...);
  END IF;
  RETURN VAR1;
 END#
```

ALTER … REGENERATE (BEGIN … END#)

REBIND PACKAGE (SET VAR1 = (SELECT ID FROM TABLE(P1 LIKE EMP)); … INSERT INTO REPORT VALUES(...);)

SQL scalar function versioning

Similar to Native SQL Stored Procedure versioning:

```
-- Create a non-inline SQL scalar function with initial version V1
CREATE FUNCTION TRYVER()
RETURNS VARCHAR(20)
VERSION V1
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version1';
```

SQL scalar function versioning

Similar to Native SQL Stored Procedure versioning:

```
-- Create a non-inline SQL scalar function with initial version V1
CREATE FUNCTION TRYVER()
RETURNS VARCHAR(20)
VERSION V1
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version1';

-- Add a second version V2 of the function created above
ALTER FUNCTION TRYVER ADD VERSION V2 ()
RETURNS VARCHAR(20)
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version2';
```

SQL scalar function versioning

Similar to Native SQL Stored Procedure versioning:

```
-- Create a non-inline SQL scalar function with initial version V1
CREATE FUNCTION TRYVER()
RETURNS VARCHAR(20)
VERSION V1
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version1';

-- Add a second version V2 of the function created above
ALTER FUNCTION TRYVER ADD VERSION V2 ()
RETURNS VARCHAR(20)
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version2';

-- Invoke the function
SELECT TRYVER() FROM SYSIBM.SYSDUMMY1;
result: Running version1
```

SQL scalar function versioning

Similar to Native SQL Stored Procedure versioning:

```
-- Create a non-inline SQL scalar function with initial version V1
CREATE FUNCTION TRYVER()
RETURNS VARCHAR(20)
VERSION V1
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version1';

-- Add a second version V2 of the function created above
ALTER FUNCTION TRYVER ADD VERSION V2 ()
RETURNS VARCHAR(20)
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version2';

-- Invoke the function
SELECT TRYVER() FROM SYSIBM.SYSDUMMY1;
result: Running version1
-- Now make V2 version the active version
ALTER FUNCTION TRYVER ACTIVATE VERSION V2;
```

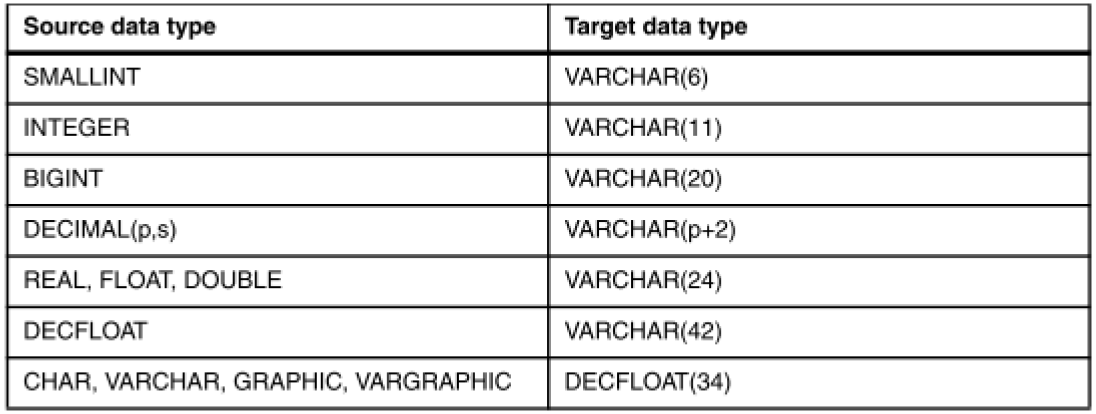
SQL scalar function versioning

Similar to Native SQL Stored Procedure versioning:

```
-- Create a non-inline SQL scalar function with initial version V1
CREATE FUNCTION TRYVER()
RETURNS VARCHAR(20)
VERSION V1
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version1';

-- Add a second version V2 of the function created above
ALTER FUNCTION TRYVER ADD VERSION V2 ()
RETURNS VARCHAR(20)
LANGUAGE SQL
DETERMINISTIC ...
RETURN 'Running version2';

-- Invoke the function
SELECT TRYVER() FROM SYSIBM.SYSDUMMY1;
result: Running version1
-- Now make V2 version the active version
ALTER FUNCTION TRYVER ACTIVATE VERSION V2;
-- Invoke the function again
SELECT TRYVER() FROM SYSIBM.SYSDUMMY1;
result: Running version2
```

Implicit cast support for strings and numerics

- Character or graphic strings and numeric data types are compatible (except for LOBs and non-Unicode graphic strings)
- DB2 can perform an implicit cast between those data types
- You can directly insert or compare the values of those data types:
  e.g.: ... WHERE EMP_ID = '100', INSERT INTO(CHARCOL) VALUES(123), etc.
- Implicit cast result data type is determined based on the following table:



Source data type	Target data type
SMALLINT	VARCHAR(6)
INTEGER	VARCHAR(11)
BIGINT	VARCHAR(20)
DECIMAL(p,s)	VARCHAR(p+2)
REAL, FLOAT, DOUBLE	VARCHAR(24)
DECFLOAT	VARCHAR(42)
CHAR, VARCHAR, GRAPHIC, VARGRAPHIC	DECFLOAT(34)

Date-time constants

DB2 10 supports ANSI/ISO SQL standard form of a datetime constant:

```
DATE '18.01.1977'                     -- date (EUR format)

TIME '15.30.00'                       -- time (EUR format)

TIMESTAMP '2007-05-14 11:55:00.1234'  -- timestamp
TIMESTAMP '2007-05-14T11:55:00.1234'  -- timestamp

TIMESTAMP '2007-05-14 11:55:00.1234+08:00'  -- timestamp with time zone UTC+8
TIMESTAMP '2007-05-14 11.55.00Z'            -- timestamp with time zone UTC


SELECT … WHERE DATECOL > '2011-01-01'       -- date column compared to string literal
SELECT … WHERE DATECOL > DATE'2011-01-01'   -- date column compared with date literal


SELECT HEX( DATE'2010-08-06' )        -- display date constant
FROM   SYSIBM.SYSDUMMY1;
-- result: 20100806                   -- i.e. DATE'2010-08-06' is a 4-byte date

SELECT HEX(     '2010-08-06' )        -- display character string constant
FROM   SYSIBM.SYSDUMMY1;
-- result: F2F0F1F060F0F860F0F6       -- i.e. '2010-08-06' is a 10-byte string
```

Extended indicator variables

There are frequent situations where an application needs to insert or update data only for a subset of columns for a given table.

The application developers are faced with a dilemma of how to provide an application that could handle all possible insert or update requests when it is not known which columns are being inserted to or updated until application execution time:

- "custom" dynamic SQL queries
- Code all combinations ahead of time
- One SQL but application must determine values for all needed columns

- New extended indicator variable values:
- -5 means DEFAULT
- -7 means UNASSIGNED (treat as if it was not specified)
- -1, -2, -3, -4, -6 means NULL

- EXTENDEDINDICATOR BIND/REBIND PACKAGE option
- WITH EXTENDED INDICATORS PREPARE attribute

Extended indicator variables

Special extended indicator variable values can be specified only for host variables (parameter markers) that appear in:

- Set assignment list of an UPDATE operation in UPDATE or MERGE statements
- The values list of an INSERT operation in INSERT or MERGE statements
- The select list of an INSERT statement in the FROM clause of the SELECT statement
- The source-table parameter of a MERGE statement

Furthermore, the host variable cannot be part of an expression other than an explicit cast (if the target column is not nullable, the explicit cast can be only to the same data type as the target column).

Extended indicator variables - Example

```
...
memset(&hv_indicators, 0, sizeof(hv_indicators));
strcpy(hv_name,    "Michael");          /* use value      */

strcpy(hv_country, "Australia");        /* use value      */

hv_indicators.hvi_name    = -7;         /* skip update    */
hv_indicators.hvi_country = -5;         /* use DEFAULT    */
hv_indicators.hvi_city    = -7;         /* skip update    */
hv_indicators.hvi_zip     = -7;         /* skip update    */
EXEC SQL
UPDATE TRYINDVAR SET
 NAME    = :hv_name:hvi_name
,COUNTRY = :hv_country:hvi_country
,CITY    = :hv_city:hvi_city
,ZIP     = :hv_zip:hvi_zip
;
```

Timestamp with TIME ZONE

New data type: TIMESTAMP WITH TIME ZONE

Timestamp plus time zone: 2010-08-06 12.30.00-8:00 (UTC-8)
(UTC is 2010-08-06 20.30.00 - subtract UTC part to convert)

Valid range for time zone:
SQL standard specification: -12:59 to +14:00
W3C XML standard for XSD schema definition specifies: -14:00 to +14:00

Because the valid range for a CURRENT TIME ZONE special register is -24:00 to +24:00, the valid range for the time zone offset was chosen to also be -24:00 to 24:00 for compatibility.

Timestamp with TIME ZONE

Determining the implicit TIME ZONE:

DSNHDECP parameter IMPLICIT_TIMEZONE and a new special register SESSION TIME ZONE aid in determining the implicit time zone.

The implicit time zone is determined as follows:

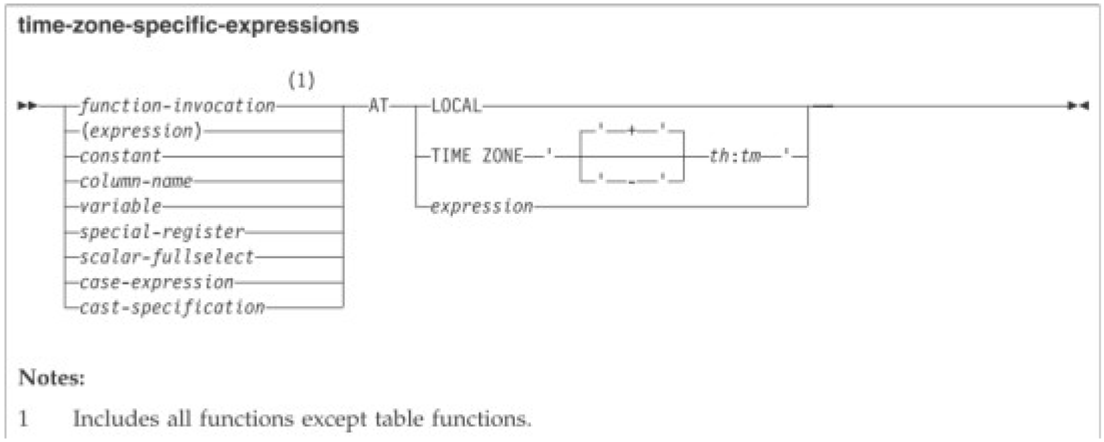
If IMPLICIT_TIMEZONE is not specified or is specified as CURRENT, the implicit time zone is the value of the CURRENT TIME ZONE special register.

If IMPLICIT_TIMEZONE is specified as SESSION, the implicit time zone is the value of the SESSION TIME ZONE special register.

If IMPLICIT_TIMEZONE is specified as a character string in the format of '±th:tm', the implicit time zone is the time zone value represented by that character string.

Time zone specific expressions

You can use time zone specific expressions to adjust timestamp values and character-string or graphic-string representations of timestamp values to specific time zones



Notes:

1 Includes all functions except table functions.

AT LOCAL – value adjusted for the local time zone using SESSION TIME ZONE special register value.

AT TIME ZONE ‘±th:tm’ - value adjusted for the specified time zone

Time zone specific expressions - Examples

```
-- establish session time zone as UTC+2
SET SESSION TIMEZONE = '+2:00';
```

Time zone specific expressions - Examples

```
SET SESSION TIMEZONE = '+2:00';

-- adjust column/literal to LOCAL
SELECT '2007-05-14-11:55:00.0 -8:00' AT LOCAL
FROM   SYSIBM.SYSDUMMY1;
-- result: 2007-05-14-21.55.00.000000+02:00
```

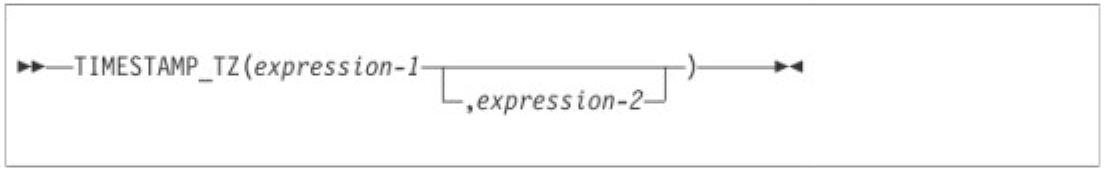
Time zone specific expressions - Examples

```
SET SESSION TIMEZONE = '+2:00';

-- adjust column/literal to LOCAL
SELECT '2007-05-14-11:55:00.0 -8:00' AT LOCAL
FROM   SYSIBM.SYSDUMMY1;
-- result: 2007-05-14-21.55.00.000000+02:00
-- how did it get calculated:
-- 1) … 11:55:00.0 -8:00 is 19:55 in UTC
-- 2) SESSION TIMEZONE has '+2:00'
--    so we add that to UTC:  19:55 + 2:00 = 21:55
```

Time zone specific expressions - Examples

```
SELECT '2007-05-14-11:55:00.0 -8:00' AT TIME ZONE '+00:00'
FROM   SYSIBM.SYSDUMMY1;
-- result: 2007-05-14-19.55.00.000000+00:00
-- this is UTC


SELECT '2007-05-14-11:55:00.0 -8:00'  AT TIME ZONE ('-'||'7'||':'||'00')
FROM   SYSIBM.SYSDUMMY1;
-- result: 2007-05-14-12.55.00.000000-07:00
```

Time zone specific scalar functions



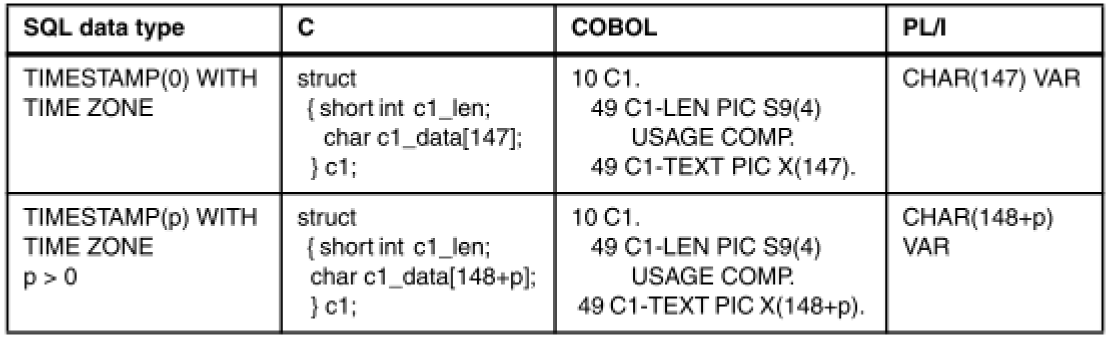
The TIMESTAMP_TZ function returns a TIMESTAMP WITH TIME ZONE value from the input arguments.

Examples (assume implicit time zone is '+8:00')

```
TIMESTAMP_TZ(TIMESTAMP '2007-05-14-12.55.00')          ==> 2007-05-14-12.55.00+08:00
TIMESTAMP_TZ(TIMESTAMP '2007-05-14-12.55.00+2:00')     ==> 2007-05-14-12.55.00+02:00
TIMESTAMP_TZ(TIMESTAMP '2007-05-14-12.55.00','-7:00') ==> 2007-05-14-12.55.00-07:00
```

Time zone – Application programming

Declarations generated by DCLGEN:



SQL data type	C	COBOL	PL/I

TIMESTAMP(0) WITH TIME ZONE	struct
	{ short int c1_len;
	char c1_data[147];
	} c1;
		10 C1.
		49 C1-LEN PIC S9(4)
		USAGE COMP.
		49 C1-TEXT PIC X(147).
			CHAR(147) VAR

TIMESTAMP(p) WITH TIME ZONE
p > 0
	struct
	{ short int c1_len;
	char c1_data[148+p];
	} c1;
		10 C1.
		49 C1-LEN PIC S9(4)
		USAGE COMP.
		49 C1-TEXT PIC X(148+p).
			CHAR(148+p)
			VAR

For Java:

```
TIMESTAMP(p) WITH TIME ZONE ==> java.sql.TimestampTZ
```

For more on SQL and Application Enablement, see “DB2 10 for z/OS Technical Overview” (SG24-7892-00)

Thank you !