IBM Information Management

IBM Informix Dynamic Server (IDS) Cheetah v11.10:

Integrated Solutions and SQL Enhancements

Author: Keshava Murthy

Architect, IBM Informix

February 20, 2007

OVERVIEW 3

INTEGRATED SOLUTIONS

NAMED PARAMETERS SUPPORT FOR JDBC 4
ENHANCED CONCURRENCY WITH COMMITTED READ LAST COMMITTED ISOLATION LEVEL . 5
IMPROVED CONCURRENCY WITH PRIVATE MEMORY CACHES FOR VIRTUAL PROCESSORS. 7
HIERARCHICAL DATA TYPE (NODE DATABLADE) 7
BASIC TEXT SEARCH INDEX 9
BINARY DATA TYPES 12
MQ MESSAGING IN IDS APPLICATIONS 14

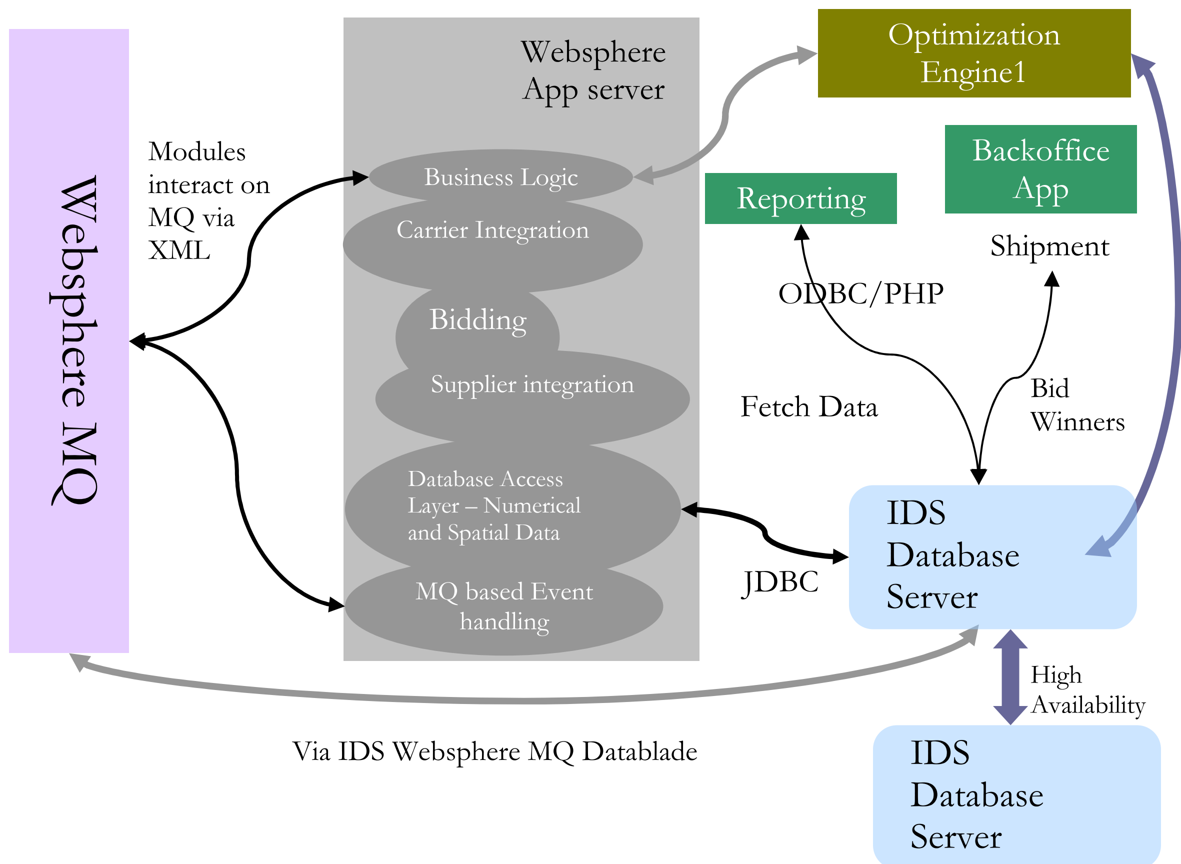
SQL ENHANCEMENTS

FULL SUPPORT FOR SUBQUERIES IN FROM CLAUSE (DERIVED TABLES OR TABLE EXPRESSIONS) 19
ENHANCEMENTS TO DISTRIBUTED QUERIES 20
INDEX SELF-JOIN ACCESS METHOD 27
OPTIMIZER DIRECTIVES IN ANSI-COMPLIANT JOINED QUERIES 29
IMPROVED STATISTICS COLLECTION AND QUERY EXPLAIN FILE. 30
XML PUBLISHING AND XPATH FUNCTIONS 33

Overview

With the introduction of the application server and then various middleware layers that can use data services, application design has gone from a simple two-tier client-server architecture, to three tier, and now to n-tier. Data services now need to include modeling and standard based query language support, high throughput with lower maintenance overhead, support for data access APIs (JDBC, .NET, etc), and integration features like messaging and XML support. IDS 11.10 Cheetah has enhanced and added features to help develop, integrate, and deploy IDS in complex integration scenarios requiring data services. This document describes some of these new Cheetah features:

- Query language support for table expressions and enhanced trigger support
- Enhanced distributed query support for extended types
- Named parameter support for the JDBC API
- New access methods (Index Self-Join), better statistics collection, and an enhanced explain file
- A new built-in text search index and a new hierarchical (Node) data type for modeling real-world hierarchies
- XML publishing functions for converting result sets into an XML document (publish functions) and for extracting portions of an XML document using an XPATH expression.
- Better concurrency with a new isolation level – committed read last committed
- MQ functionality built into IDS 10.00.xC3 to help heterogeneous integration and SOA enablement



Integrated Solutions

Named Parameters support for JDBC

Cheetah supports Named Parameters to make JDBC application code easier to write and more readable. Each function parameter has a name and datatype. Prior to this release, JDBC could only bind values to these parameters by their position in the parameter signature. In Cheetah, parameter binding can happen by the implicit position of the function signature or by explicitly naming the parameter and value.

```
CREATE FUNCTION order_item(cust_id int,
               item_id int,
               count int,
               billing_addr varchar(64),
               billing_zip int,
               shipping_addr varchar(64),
               shipping_zip int) return int status;

INSERT INTO order_tab(cust_id, item_id, count);
INSERT INTO billing_tab(cust_id, , billing_addr, billing_zip);
INSERT INTO shipping_tab(cust_id, , shipping_addr, shipping_zip);

Return 1;
END FUNCTION;
```

There are two ways to bind values to parameters:

1. implicit positional binding:

```
execute function order_item(5739, 8294, 5, "4100 Bohannon Dr.", 94025,
                         "345, university ave.", 94303);
```

2. explicit parameter naming:

```
execute function order_item(cust_id=5739,
                         count=5,
                         item_id=8294,
                         shipping_addr="345, University ave.",
                         shipping_zip=94303,
                         billing_addr="4100 Bohannon Dr.",
                         billing_zip=94025);
```

In the first case, you have to look up the signature to find out the values to parameter mapping; it's easier when you specify the parameter names during invocation. The advantage of named parameters becomes obvious when you see its usage in a JDBC program.

Example of implicit positional binding:

```
CallableStatement cstmt = con.prepareCall("call order_item(?, ?, ?, ?, ?, ?,
?)");
```

```
// Set parameters (positional notation)
cstmt.setInt(1, 5739 );
cstmt.setInt(2, 8294);
cstmt.setString(6, "345, University ave.");
cstmt.setInt(7, 94303);
cstmt.setString(4,"4100 Bohannon Dr.");
cstmt.setInt(5, 94025);
cstmt.setInt(3,5);

// Execute
cstmt.execute();
```

Named parameters help you avoid looking up the function signature while writing code, which makes it less error prone and makes the code easier to read. Here's the same code rewritten using named parameter notation:

```
// Set parameters (named notation)
cstmt.setInt("cust_id", 5739 );
cstmt.setInt("item_id", 8294);
cstmt.setString("shipping_addr", "345, University ave.");
cstmt.setInt("shipping_zip", 94303);
cstmt.setString("billing_addr","4100 Bohannon Dr.");
cstmt.setInt("billing_zip", 94025);
cstmt.setInt("count", 5);

// Execute
cstmt.execute();
```

Named notation self-documents the code through the obvious assignment of the parameter and its value. Moreover, you can define the used parameters in any order and omit parameters that have default values.

Enhanced Concurrency with Committed Read Last Committed Isolation level

Multi-user database systems implement ACID (Atomicity, Consistency, Isolation and Durability) properties by locking the rows that were updated, or by locking the rows applications request and then serializing access to data that multiple transactions are interested in. These locking protocols come with drawbacks of that include lost concurrency and throughput, lock conflicts and deadlocks, lock maintenance overhead, and so on.

For example, any modification to a table row can result in a lock being held on the row until the modifying transaction commits or rolls back. Readers for that row under any of the existing isolation levels except "Dirty Read" would need to block until the modifying transaction is committed or rolled back. Similarly, a reader executing in the "Repeatable Read" isolation level would lock the row for the duration of the transaction, preventing updates to that row by anyone else.

This feature implements a form of multi-versioning, where readers can be returned one of two versions: the "last committed version" of the data, or the "latest" data. The two versions would be the same under the existing lock-based isolation levels of Committed Read, Cursor Stability, and Repeatable Read, while they can be different for "Dirty Read" isolation level and for the new "Last Committed" option.

The “Last Committed” option is an optional extension to the existing Committed-Read isolation level.

Syntax:

```
Set Isolation To Committed Read Last Committed;
```

Example scenario: Transfer $400.00 from account number 1234 to number 3456. We compare the different isolation levels to show the values you get and the wait times.

Time	Transaction1	Transaction2 COMMITTED READ (default in logged database)	Transaction3 DIRTY READ	Transaction4 LAST COMMITTED (New in IDS 11.10)
1.	-- Current balance of customer 1234 is 1250.00			
2.	set isolation to read committed;			
3.	begin work;			
4.	update cust_tab set balance =balance – 400 where cust_id = 1234;	begin work;	begin work;	begin work;
5.	-- balance of customer 1234 is 850.00			
6.	update cust_tab set balance = balance + 400 where cust_id = 3456;	select balance from cust_tab where custid = 1234; -- wait for the lock on row for customer 1234	select balance from cust_tab where custid = 1234; -- No waiting. will return 850.00.	select balance from cust_tab where custid = 1234; -- No waiting -- will return 1250.00
7.	insert into daily_tab(“transfer”, 1234, 3456, 400);	-- Status: lock wait	-- Continue processing	-- Continue processing
8.	Commit work;	-- Status: lock wait	-- do more	-- do more
9.		--will return 850.00	-- do more	-- do more

To avoid waiting on the row lock, applications previously used “Dirty Read” isolation level. This isolation level provides dirty information for the row. In the example above, retrieving the dirty value of customer 1234 can be unreliable because the transfer might not be successful and the transaction could be rolled back. The new isolation level, “Committed Read Last Committed,” provides the values the row had before the latest update from an uncommitted transaction -- in this case Transaction1 modified the row for the customer ID 1234. That transaction is not guaranteed to commit or rollback. If an application needs to retrieve the value from the previously committed row, it can now use the “Committed Read Last Committed” isolation to enforce this isolation on the data it retrieves.

For application development debugging and support scenarios, the new SET ENVIRONMENT statement overrides the isolation mode for the current session when the user sets the "Committed Read Last Committed" isolation level:

```
SET ENVIRONMENT USELASTCOMMITTED ["None" | "Committed Read" | "Dirty Read" |
"All" ]
```

The configuration parameter USELASTCOMMITTED can be set to None, 'Committed Read', 'Dirty Read' or All to permanently override the "Committed Read Last Committed" isolation level.

Improved Concurrency with Private Memory Caches for Virtual Processors.

This feature adds an optional private memory cache for each CPU VP. This new cache contains blocks of free memory from 1 to 32 blocks in length. Each memory block is 4096 bytes. The purpose of this cache is to speed access to memory blocks. Normally the server performs a bitmap search to find memory blocks of a requested length; this search requires a lock and can affect concurrency when large number of CPU VPs are configured on a multiprocessor system.

This feature can be enabled by setting the VP_MEMORY_CACHE_KB configuration parameter in the onconfig file. Setting it to 0 turns the private memory cache off. The minimum value for this parameter is 800; at least 800 kilobytes must be allocated to each CPU VP. The memory used can be calculated by multiplying this value by the number of CPU VPs: this should not exceed 40% of the memory limit as specified in the SHMTOTAL configuration parameter. The cache size should, in most circumstances, be set much smaller than the maximum value; this will result in a high cache hit ratio and not bind memory blocks to a single CPU VP. In addition, the normal memory management routines are capable of combining adjacent free blocks of memory whereas the CPU VP memory cache does not combine adjacent free memory blocks.

This feature can be turned on and off while the server is running with the onmode command:

```
onmode -wm VP_MEMORY_CACHE_KB=<value>  -- Configure current value
onmode -wf VP_MEMORY_CACHE_KB=<value>  -- modify in $ONCONFIG
```

The onmode -wf command updates the onconfig file so that the new value is used when you restart the server. If you set this value to 0 by using the onmode command, then the memory caches are emptied and the feature is disabled. You can monitor the private cache by using the onstat -g vpcache command.

Hierarchical Data Type (Node DataBlade)

Hierarchical relationships can be illustrated by these examples: John Smith is the father of Paul, and Paul is the father of Peter; therefore, John is an ancestor of Peter by transitivity. A crate of cola packages contains many cases of cola which in turn contains many six-packs of cola cans. Each of these entities – crates, cases, six-packs, cans – will have product identifiers. When you design the schema to store such hierarchical information, it is not enough to store the values, it's also important to model the hierarchical relationships and enable queries on relationships among the entities in the hierarchy. You want to answer questions like which crate did a can of cola came from so you can determine from which bottling facility this cola can came. Relational databases are very good at handling relations and logical hierarchies. However, if you attempt to model hierarchical data and

try to implement transitive closure on multi-level hierarchies, such as process dependency, with the base RDBMS features, you need to write very convoluted SQL.

IDS 11.10 introduces a new node data type to help you easily model hierarchical relationships. The data type and the support functions – ancestor, depth, getparent, getmember, length, etc. – are packaged as the Node DataBlade module, version 2.0. You need to register this datablade in your database before you can use the functionality.

The node data type is an opaque data type that models the tree structure rather than reducing hierarchies to overly simple relations. Each value represents the edge of the hierarchy, not simply a number or a string. Therefore, when you increment node 1.9, you get node 1.10, and not the numerical incremental value of 1.91 or 2.9. For example:

```
CREATE TABLE Employees(Employee_Id NODE, desc VARCHAR(60));

INSERT INTO Employees VALUES ('1.0',     "CEO");
INSERT INTO Employees VALUES ('1.1',     "VP1");
INSERT INTO Employees VALUES ('1.1.1',   "Admin for VP1");
INSERT INTO Employees VALUES ('1.2',     "VP2");
INSERT INTO Employees VALUES ('1.2.1',   "Manager1");
INSERT INTO Employees VALUES ('1.2.2',   "Manager2");
INSERT INTO Employees VALUES ('1.2.2.1', "Admin for Manager2");

-- Retrieve the hierarchy for the "Admin for Manager2"
SELECT *
FROM Employees E
WHERE isAncestor(Employee_Id, '1.2.2.1')
ORDER BY E.Employee_Id ;


employee_id       desc

1.0               CEO
1.2               VP2
1.2.2             Manager2

-- return all the people under a manager, in this example under VP2
SELECT *
FROM   Employees E
WHERE  Employee_Id > '1.2';

employee_id        desc

1.2.1              Manager1
1.2.2              Manager2
1.2.2.1            Admin for manager2


-- retrieve list each employee and their position in order.
select e.employee_id, e.desc from employees e order by depth(e.employee_id)
desc;


employee_id         desc

1.2.2.1             Admin for Manager2
```

```
1.1.1               Admin for VP1
1.2.1               Manager1
1.2.2               Manager2
1.1                 VP1
1.2                 VP2
1.0                 CEO


-- generate the hierarchy, bottom up.
select e.employee_id,
       e.desc,
       depth(e.employee_id) level,
       getparent(e.employee_id) manager_id
from employees e
order by 3 desc, 4

employee_id     desc                level       manager_id

1.2.2.1        Admin for Manager2     4            1.2.2
1.1.1          Admin for VP1          3            1.1
1.2.2          Manager2               3            1.2
1.2.1          Manager1               3            1.2
1.1            VP1                    2            1.0
1.2            VP2                    2            1.0
1.0            CEO                    1
```

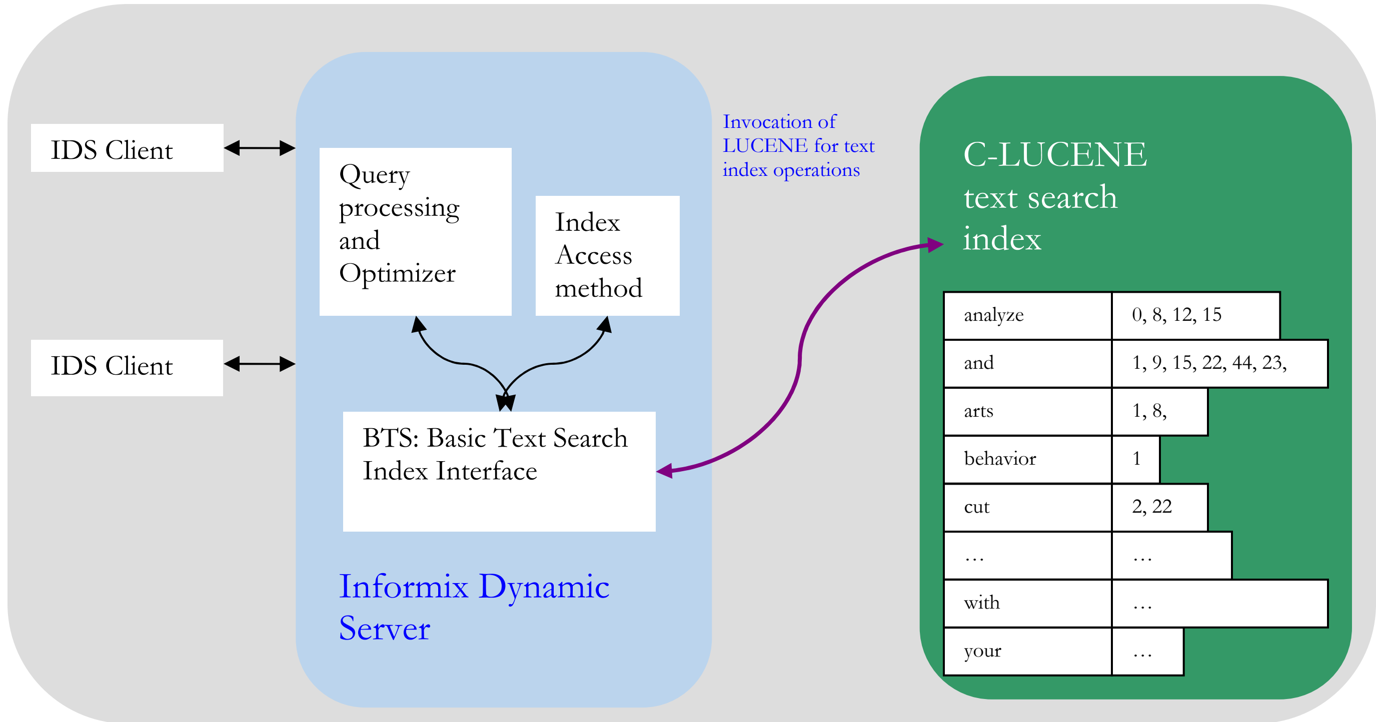
See the IDS Cheetah documentation and the developerWorks article on the Node DataBlade module at http://www-128.ibm.com/developerworks/db2/zones/informix/library/techarticle/db_node.html.

Basic Text Search Index

The Basic Text Search DataBlade module (BTS) provides simple word and phrase searching on an unstructured document repository. This document repository is stored in a column of a table. The column can be of type char, varchar, nvarchar, lvarchar, BLOB, or CLOB. IDS nvarchar can store multibyte strings and this text search engine can index and search the nvarchar columns. The search predicate can be a simple word, a phrase, a simple Boolean operator (AND, OR, or NOT), single or multiple wildcard searches, a fuzzy search, or a proximity search. The search engine is provided by the open source CLucene text search package.

The index works in DIRTY READ isolation level regardless of the isolation level set in the server. This implies that any modification – INSERT, UPDATE, DELETE – performed on the index by transaction T1 will be immediately seen by any other transaction accessing the index after the modification and before T1 is committed (or rolled back). Notice that updates to the index are synchronous with table updates.

The following illustration shows how BTS and CLucene work together.



Usage:

```
-- First, register bts.1.0 into your database using blademgr
mkdir /work/myapp/bts_expspace_directory
-- Create an external space to hold the index
onspaces -c -x bts_extspace -l /work/myappbts_expspace_directory

--Create a table with a BTS index
CREATE TABLE article_tab(id integer, title lvarchar(512));


-- Load the data below.
```

id (integer)	title (lvarchar(512))
0	Understanding locking behavior and analyze lock conflicts in IDS
1	Informix and Open source: database defense against the dark political arts
2	Cut out the Middle-Man: Use Informix with J/Foundation to host a Java application service
3	Optimize your BAR performance using parallel backups on IDS
5	Flexible fragmentation strategy in Informix Dynamic Server 10.00
6	Push the limits of Java UDRs in Informix Dynamic server 10.00
…	…

```
CREATE INDEX title_index ON article_tab(title bts_lvarchar_ops)
      USING bts in bts_extspace;
```

Examples:

```
select id from article_tab where bts_contains(title, 'informix')
```

Results:

```
   id

    5
    1
    6
    2
```

Plan:

```
Estimated Cost: 0
Estimated # of Rows Returned: 1

  1) keshav.article_tab: INDEX PATH

    (1) VII Index Keys: title   (Serial, fragments: ALL)
        VII Index Filter:
informix.bts_contains(keshav.article_tab.title,informix )


SELECT id FROM article_tab
WHERE bts_contains(title, ' "use informix" ');

        id

          2

1 row(s) retrieved.


-- with the AND Boolean operator (&& and + are allowed as well)
SELECT id FROM article_tab WHERE bts_contains (title, 'informix AND
dynamic') ;

        id

          5
          6

2 row(s) retrieved.
```

```
-- with the single character wildcard
SELECT id FROM article_tab WHERE bts_contains (title, 'inf*rmix') ;

        id

          5
          1
          6
          2

4 row(s) retrieved.

-- with the proximity search
SELECT id FROM article_tab WHERE bts_contains (title,'"java"~10') ;

        id

          6
          2

2 row(s) retrieved.
```

Binary Data Types

In prior releases of IDS, binary data could be stored in BYTE or BLOB datatypes. Both of these types are designed to handle large datasets – BYTE up to 2GB, and BLOB up to 4TB, are stored out of row and do not support indices on the binary data they store. For applications using smaller binary strings, IDS v11.10 has two new types to consider for this situation: binary18 and binaryvar. The input to these data types are ASCII strings with hexadecimal [0-9A-Fa-f] digits. Because the data is stored as bytes, the input hexadecimal string should have an even number of digits. Binary18 is a fixed 18bytes long type and binaryvar is a variable length and can store 255 bytes. IDS stores these types in-row and supports B-TREE indices on these types. Available indices on these binary types will be considered by the optimizer as a viable access path. These binary data types are provided with the Binary DataBlade module. After registering the Binary DataBlade module with BladeManager, you create tables using the new types:

```
create table bin18_test (int_col integer, bdt_col binary18);

insert into bin18_test values (0, '0102');
insert into bin18_test values (1, '01020304');
insert into bin18_test values (2, '0102030405060708');
insert into bin18_test values (3, '0102030405060708090A0B0C');

-- The hexadecimal prefix 0x is allowed as a prefix.
insert into bin18_test values (3, '0X0102030405060708090A0B0C');

create table bindata_test (int_col integer, bin_col binaryvar)

insert into bindata_test values (1, '30313233343536373839')
insert into bindata_test values (2, '0X30313233343536373839')

-- create indices on binary types
create index idx_bin18 on bin18_test(bdt_col);
create index idx_binvar on bindata(bin_col);
```

Both data types are indexable with a B-tree index and hence are considered during optimization of the query. IDS does not create distributions on binary columns and the optimizer assumes a selectivity of 0.1 from the binary column. Use query directives to force the selection of alternative access paths when you find these assumptions result in sub optimal plans for queries with binary data types.

Following new functions operate on binary columns:

- length(binary_column)
- octet_length(binary_column)

Example:

```
select length(bin_col) from bindata_test where int_col=1;

(expression)

          10
```

The following bitwise operations are supported on binary datatypes:

- bit_and(arg1, arg2): implements the bitwise AND operator
- bit_or(arg1, arg2): implements the bitwise OR operator
- bit_xor(arg1, arg2): implements the bitwise XOR (exclusive OR) operator
- bit_complement(arg1): implements the bitwise NOT

Example:

```
create table bindata_test (int_col integer, bin_col binaryvar)

insert into bindata_test values (1, '00001000');
insert into bindata_test values (2, '00002000');
insert into bindata_test values (3, '00004000');
insert into bindata_test values (4, '023A2DE4');

select bit_or(bin_col, '00004000')
from bindata_test where int_col=2;  -- add CASE stmt here.


(expression)  00006000

select bit_and(bit_or(bin_col, '40404040'), '01010200')
from bindata_test where int_col=2;

(expression)  00000000

select bit_complement(bin_col) from bindata_test where int_col=4;

(expression)  FDC5D21B

select bit_xor(bin_col,'00004040') from bindata_test where int_col=3;

(expression)  00000040
```

If either argument is NULL, a NULL binary is returned. If the lengths of the two arguments are different, the operation is performed up to the length of the shorter string. The rest of the data from the longer binary string is appended to the result of the binary operation thus far.

.Example:

```
create table x(a binaryvar, b binaryvar);
insert into x values('aa', 'aaaa');

select bit_and(a,b), bit_or(a,b), bit_xor(a,b), bit_complement(a)
from x;

(expression)  AAAA
(expression)  AAAA
(expression)  00AA
(expression)  55
```

MQ Messaging in IDS Applications

Websphere MQ (WMQ) provides reliable messaging for distributed, heterogeneous applications to interact, exchange information, delegate jobs, and offer services by action upon information received. Historically, applications built for this scenario had to be written with custom code, and had to manage multiple connections and route data between WMQ and IDS. IDS 10.00.UC3 introduced built-in support to enable IDS applications to interact with WMQ via SQL, thus eliminating the overhead. Subsequent releases included enhanced and extended platform support for WMQ. MQ functions are provided in IDS as the MQ DataBlade module.

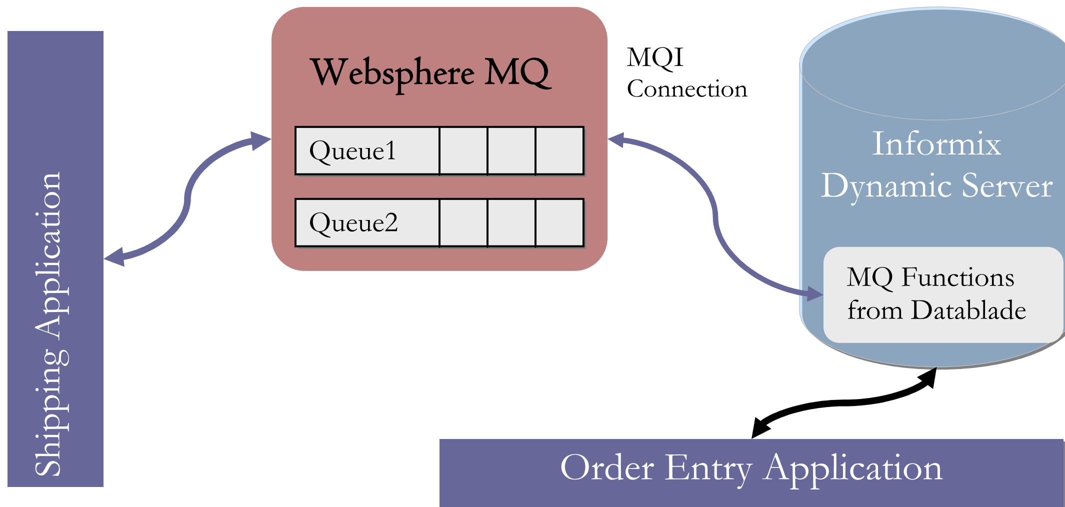
Whenever you buy a book on amazon.com or enroll in e-business with ibm.com, the order event triggers a work flow of the information through multiple modules: user account management, billing, packaging and shipping, procurement, customer service, and partner services. The execution in triggered modules generates subsequent work flow. To meet reliability and scaling requirements, it's typical to have application modules on multiple machines.

If you're using the same software on all systems, for example the SAP stack, the software itself usually comes with workflow management features. If the modules are running in a homogeneous environment -- for example, Linux® machines, running WebSphere and Informix -- it's easier to change information using distributed queries or enterprise replication. On the other hand, if the application is running on heterogeneous systems -- such as combinations of WebSphere, DB2®, Oracle, and Informix -- programming and setup of distributed queries or replication becomes complex and in many cases won't meet application requirements. WebSphere MQ is designed to address integration issues like this. It prefers no platform and enforces no paradigms: WebSphere MQ supports more than 80 platforms, and APIs in C, C++, Java™, Java Message Service (JMS), and Visual Basic. WebSphere MQ is also the mainstay for designing enterprise service bus (ESB) for Service Oriented Architecture (SOA).

WebSphere MQ provides a reliable store-and-forward mechanism so each module can send and receive messages to and from it. WebSphere MQ achieves this by persistent queues and APIs for programming. In addition, WebSphere MQ Message Broker -- another product in the WebSphere MQ product suite -- provides message routing and translation services. Simplicity of infrastructure means the applications must establish, for example, message formats and queue attributes. WebSphere MQ also supports publish and subscribe semantics

for queues, making it easy to send a single message to multiple receivers and subscribing messages from queues by need, similar to mailing lists.

The following illustration shows how an IDS applications can use Websphere MQ.



IDS provides SQL-callable functions to read, subscribe, and publish with MQ. These SQL-callable functions expose MQ features to IDS application and integrate the MQ operations into IDS transactions. For example, IDS uses two-phase commit protocol in distributed transactions; MQ operations commit and rollback along with IDS transactions.

Using IDS MQ functionality, sending and receiving a message to and from an MQ Queue is simple:

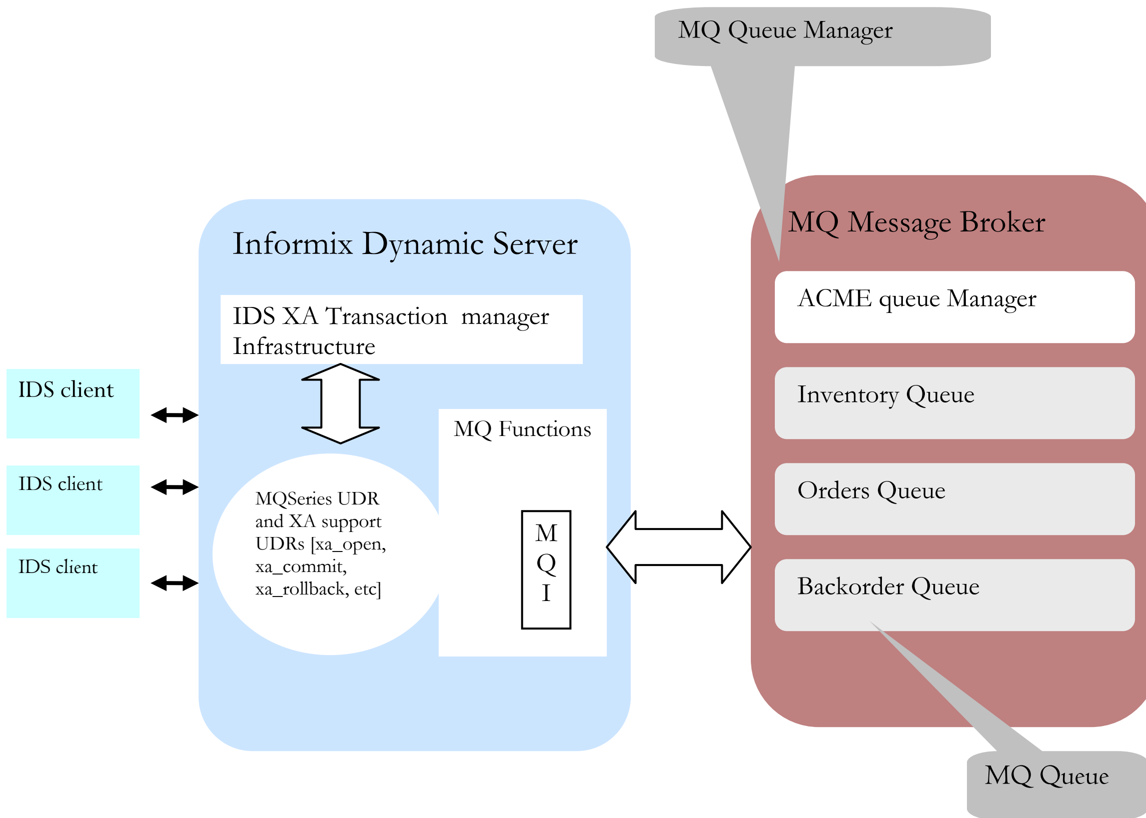
```
SELECT MQSend("CreditProc", customerid || ":" || address ||  ":"
                  || product ":" || orderid)
FROM   order_tab
WHERE customerid = 1234;

INSERT into shipping_tab(shipping_msg) values(MQReceive());

CREATE PROCEDURE get_my_order();
      define  cust_msg lvarchar[2048];
      define  customerid  char[12];
      define  address     char[64];
      define  product     char[12];

      -- Get the order from Order entry application.
      EXECUTE FUNCTION MQReceive("OrderQueue") into cust_msg;
      LET customerid = substr(cust_msg, 1, 12);
      LET address    = substr(cust_msg, 14, 77);
      LET product    = substr(cust_msg, 79, 90);

      INSERT into shipping_table(custid, addr, productid, orderid)
                  Values(customerid, address, product, :orderid);
```

```
        -- send the status to CRM application
        EXECUTE FUNCTION MQSend("CRMQueue",
                                :ordereid || ":IN-SHIPPING");
        RETURN 1;
END FUNCTION;
```

The following table lists the MQ functions in IDS.

Function Name	**Description**
MQSend()	Send a string message to a queue
MQSendClob()	Send CLOB data to a queue
MQRead()	Read a string message in the queue into IDS without removing it from the queue
MQReadClob()	Read a CLOB in the queue into IDS without removing it from the queue
MQReceive()	Receive a string message in the queue into IDS and remove it from the queue
MQReceiveClob()	Receive a CLOB in the queue into IDS and remove it from the queue
MQSubscribe()	Subscribe to a Topic
MQUnSubscribe()	UnSubscribe from a previously subscribed topic
MQPublish()()	Publish a message into a topic
MQPublishClob()	Publish a CLOB into a topic
CreateMQVTIRead()	Create a read VTI table and map it to a queue.
CreateMQVTIReceive()	Create a receive VTI table and map it to a queue.
MQTrace()	Trace the execution of MQ Functions
MQVersion()	Get the version of MQ Functions

The following illustration shows how IDS and MQ manage transactions.



When you invoke any MQ function exchanging message with MQ, you must be in a transaction, implicitly or explicitly. Transactions are necessary to provide reliable interaction between IDS and MQ. When the commit is successful, the application requires that all changes to data at IDS and MQ are persistent. When the application rolls back a transaction, any operations at MQ and on IDS are both rolled back . IDS implicitly starts a transaction when you issue DML (UPDATE, DELETE, INSERT or SELECT) and DDL statements (CREATE statements). You can explicitly start a new transaction with BEGIN WORK statements or use APIs like JDBC start a new transaction when you set autocommit to off. Note that the EXECUTE FUNCTION and EXECUTE PROCEDURE statements do not start a transaction, so you need to start a transaction before invoking an MQ function in an EXECUTE statement.

The transaction management is transparent to an application. The application uses MQ functionality under a transaction and IDS handles the commit or rollback coordination between IDS and MQ using the open two-phase commit protocol. This process is integrated into the IDS transaction manager; IDS handles MQ along with its distributed transactions involving other IDS instances. During IDS-MQ interaction, IDS opens a connection to MQ and when the application invokes the first MQ function within a transaction, IDS begins a corresponding transaction at MQ. During commit or rollback, the IDS transaction manager is aware of MQ participation in the transaction and co-ordinates the transaction with it.

The following table shows on which platforms IDS supports MQ.

IDS Version	Supported Platforms	Websphere MQ Version
10.00.xC3 and Higher	• Solaris – 32 bit • HP/UX (PA-RISC) – 32 bit • AIX – 32bit • Windows – 32bit.	Needs v5.3 and higher
10.00.xC4 and Higher	• AIX – 64bit • HP/UX (PA-RISC) – 64 bit	Needs v6.0 and higher
10.00.xC5 and Higher	• Linux (Intel) – 32 bit • Linux (pSeries) – 64bit • Solaris – 64 bit	Needs v6.0 and higher

IDS MQ functionality eliminates need for custom code development for IDS applications interacting with MQ. After you setup the queues, services, and policies, developers can use MQ functions like other built-in functions in the development environment of their choice. If you set up the READ and RECEIVE tables, developers can directly query and insert data into them using SQL statements.

SQL Enhancements

Overview

SQL Enhancements in IDS 11.10 cover a spectrum of features.

- Query language enhancements for subqueries in a FROM clause, pagination features to retrieve a window of rows from the result set
- Better trigger support – multiple triggers for the same event, a new way to access and modify trigger correlated variables from procedures called from a trigger action
- Automatic creation of distribution for the leading key of the index during create index operation and better collection of table statistics
- Explain file now has optimizer cardinality estimates and actual cardinalities along with the query plan.
- A new SAMPLING SIZE clause in the UPDATE STATISTICS statement provides better control over the number of rows to sample while creating the distribution
- A new Index Self Join access method which uses composite indices in even more situations than before.
- Distributed queries across IDS 11.10 servers support lvarchar, boolean and distinct types of all basic SQL types, lvarchar, boolean types.
- XML publishing functions to convert result sets into an XML document (publish functions) and functions which can extract portions of XML for a given XPATH expression.

Full support for subqueries in FROM clause (derived tables or table expressions)

The table reference in the FROM clause of a SELECT statement can be a table, a view, a table function (iterator functions), or collection derived tables (CDT). Result sets from CDTs and table functions are treated as transient table within the scope of the query – this transient is referred to as a derived table or a table expression. For example:

```
SELECT *
FROM (SELECT tab1.a, tab2.x, tab2.y
        FROM tab1, tab2
        WHERE tab1.a = tab2.z
        ORDER BY tab1.b ) vt(va, vb, vc),
      emptab
WHERE  vt.va = emptab.id;
```

This approach is more flexible than creating views or temporary tables, inserting data into them and using them in your query. These subqueries in the FROM clause can do most things a normal query can does, including aggregation, pagination, sorting, and grouping. The IDS optimizer treats these queries similar to queries in views. It tries to fold the subquery into the parent query without changing the meaning or affecting the output. Failing that, the result set is materialized into a temporary table. Once you've constructed the subquery and its table and column reference, you can use it in all syntactical constructs: ANSI joins, UNION, UNION ALL, etc. In prior releases, this required using a collection derived table (CDT), which required TABLE and MULTISET keywords. This feature enables you to write and generate SQL-standard compliant syntax.

Examples:

```
-- CDT syntax
SELECT SUM(VC1) AS SUM_VC1, VC2
FROM TABLE (MULTISET(SELECT C1, C2 FROM T1 )) AS VTAB(VC1, VC2)
GROUP BY VC2;

 -- New syntax
SELECT SUM(VC1) AS SUM_VC1, VC2
FROM (SELECT C1, C2 FROM T1 ) AS VTAB(VC1, VC2)
GROUP BY VC2;

-- New syntax
SELECT * FROM
( (SELECT C1,C2 FROM T3) AS VT3(V31,V32)
 LEFT OUTER JOIN
        ( (SELECT C1,C2 FROM T1) AS VT1(VC1,VC2)
        LEFT OUTER JOIN
        (SELECT C1,C2 FROM T2) AS VT2(VC3,VC4)
        ON VT1.VC1 = VT2.VC3)
ON VT3.V31 = VT2.VC3);

  -- New syntax
  SELECT *
  FROM table(foo(5)) AS vt(a), tab1 t
  WHERE vt.a = t.x;
```

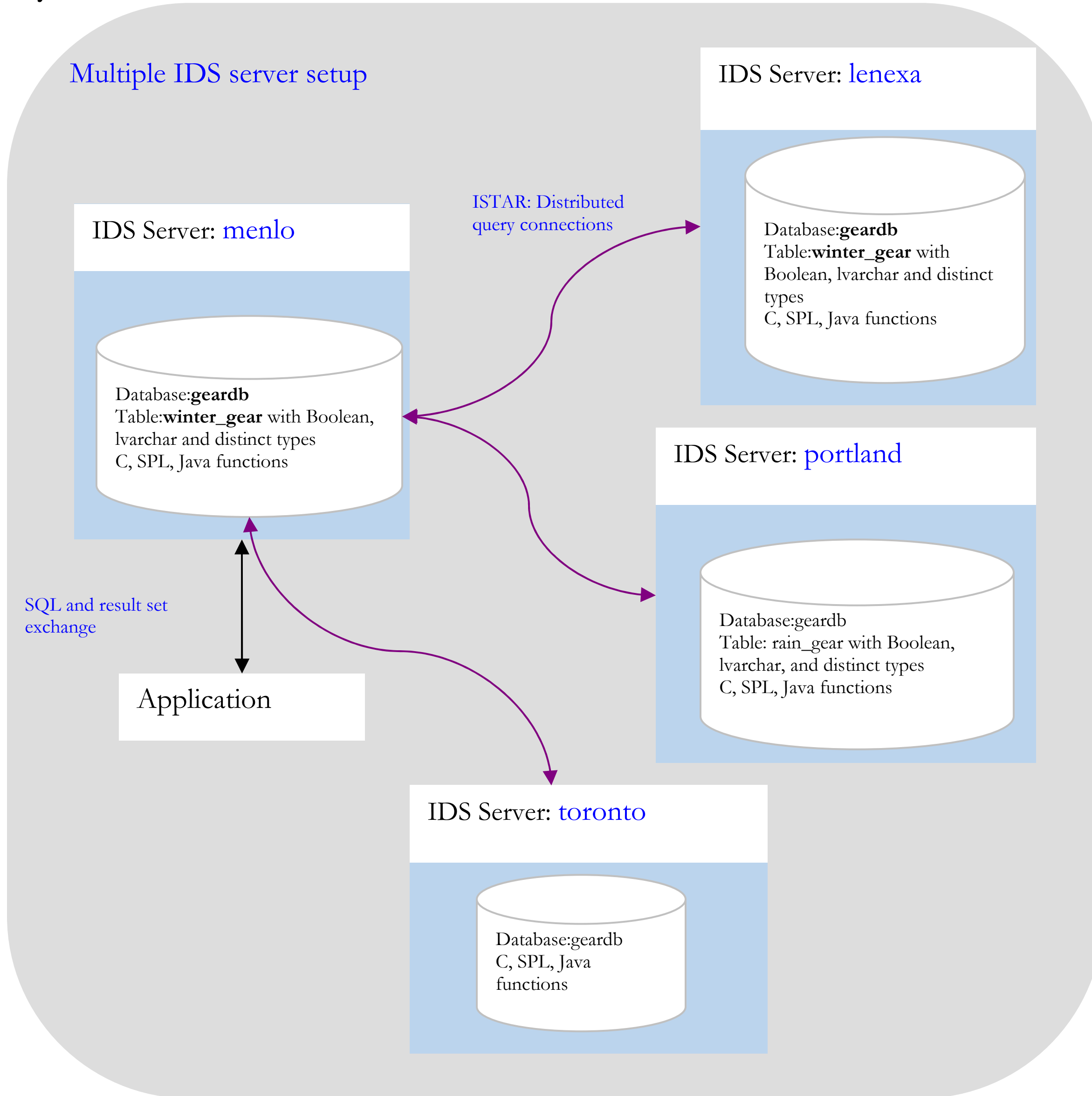
Enhancements to Distributed Queries

Each IDS database server can have multiple databases. You can have multiple IDS instances, each with multiple databases. A query across multiple databases on the same IDS instance is called a cross-database query. A query across databases from multiple IDS instances is called a cross-server or distributed query. IDS v10.00 added cross-database query support for extended types (Boolean, lvarchar, BLOB, CLOB), distinct types, and UDTs that are explicit casts of one of the supported types.

IDS v11.10 supports Boolean, lvarchar, and distinct types of all basic types, lvarchar, and Boolean types in distributed queries. IDS v11.10 also supports C and Java UDRs, in addition to SPL procedures, in distributed queries.

Consider the following IDS setup.

Example: Querying for Boolean and lvarchar data types in a distributed query



Connect to portland:

```
CREATE TABLE rain_gear(partner  int, active Boolean, desc lvarchar(4096));
```

Connect to lenexa:

```
CREATE TABLE winter_gear(partner  int, active Boolean, desc lvarchar(4096));
```

Connect to Menlo:

```
CREATE TABLE sunny_gear(partner int, active Boolean, desc
lvarchar(4096));
```

```
-- Select active partners at (Menlo, Lenexa and Portland)
SELECT x.partner, x.desc, y.desc, z.desc
FROM geardb@menlo:sunny_gear x,
     geardb@portland:rain_gear y,
     geardb@lenexa:winter_gear z
WHERE x.partner = y.partner and
      x.partner = z.partner and
      x.active = 'T' and
      y.active = 'T' and
      z.active = 'T'
ORDER BY x.partner;
```

Example: Distributed queries using distinct types (distinct types of basic types, Boolean, or lvarchar).

Connect to members@portland:

```
CREATE distinct type pound as float;
CREATE distinct type kilos as float;

CREATE function ret_kilo(p pound) returns kilos;
define x float;
let x = p::float / 2.2;
return x::kilos;
end function;

CREATE function ret_pound(k kilos) returns pound;
define x float;
let x = k::float * 2.2;
return x::pound;
end function;

create implicit cast (pound as kilos with ret_kilo);
create explicit cast (kilos as pound with ret_pound);

CREATE table members (age int, name varchar(128), weight pound);
```

Connect to members@Toronto:

```
CREATE distinct type pound as float;
CREATE distinct type kilos as float;

CREATE function ret_kilo(p pound) returns kilos;
define x float;
let x = p::float / 2.2;
return x::kilos;
end function;

CREATE function ret_pound(k kilos) returns pound;
define x float;
let x = k::float * 2.2;
return x::pound;
end function;

create implicit cast (pound as kilos with ret_kilo);
create explicit cast (kilos as pound with ret_pound);
```

```
CREATE table members (age int, name varchar(128), weight kilos);
```

Connect to members@Menlo:

```
CREATE distinct type pound as float;
CREATE distinct type kilos as float;

CREATE function ret_kilo(p pound) returns kilos;
define x float;
let x = p::float / 2.2;
return x::kilos;
end function;

CREATE function ret_pound(k kilos) returns pound;
define x float;
let x = k::float * 2.2;
return x::pound;
end function;

create implicit cast (pound as kilos with ret_kilo);
create explicit cast (kilos as pound with ret_pound);

-- select the Portland members weighing less than  Toronto members of
the same age.
select x.name, x.weight
FROM members@portland:members x
       members@toronto:members y
WHERE x.age = y.age and
      x.weight < y.weight    -- compares pounds to kilos.
group by x.weight;
```

Similarly, distributed queries in Cheetah can invoke C and Java user defined routines in remote database servers. Prior versions allowed just SPL procedure invocation across database servers. Since Cheetah has enabled the use of lvarchar, boolean and distinct types in cross server queries, these types can be used in parameters and return values in cross server invocation.

From Menlo:

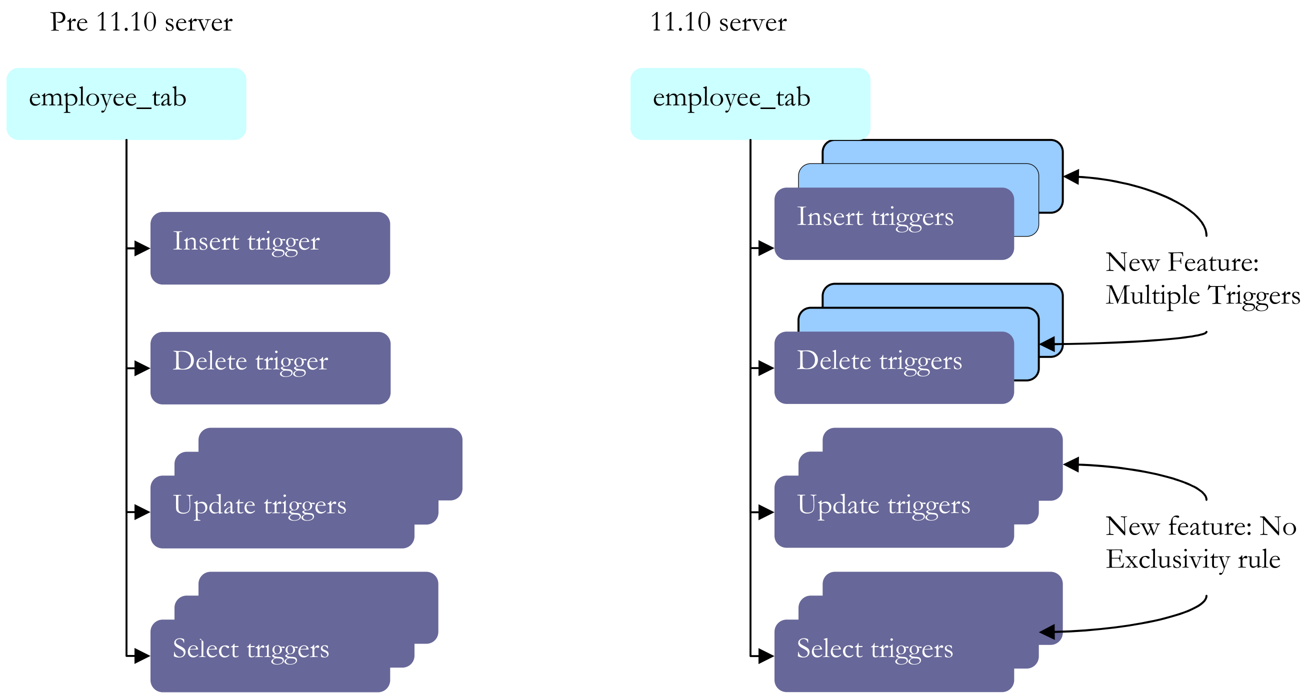
```
Select x.name, x.weight, x.height,
members@portland:body_mass_index(x.weight, x.height) as bmi        FROM
members@portland:members x WHERE x.id = 1234;
```

Trigger Enhancements

A trigger is a database mechanism that automatically executes a set of SQL statements when a certain event occurs. Cheetah enhances two aspects of triggers: multiple triggers for the same event, and a new type of user-defined routine, called a trigger routine, which can be invoked from the FOR EACH ROW section of the triggered action. These SPL routines can apply procedural logic of the SPL language in operations on OLD and NEW column values in the rows processed by the triggered actions.

Multiple INSERT and DELETE triggers:

Triggers are sets of SQL statements executed when a specific SQL operation – INSERT, UPDATE, DELETE or SELECT – is performed on a table or a view. In prior releases, one INSERT and DELETE trigger could be created per table; UPDATE and SELECT triggers could be for a complete table or selected columns, and only one trigger could be defined on a column. Cheetah allows creation of multiple INSERT and DELETE triggers on the same table and allows creation of multiple UPDATE and SELECT triggers on the same column. For each event, all available and applicable triggers are executed without a predetermined order.



Single insert and delete triggers, and multiple update and select triggers on mutually exclusive columns

Multiple insert, update, delete and select triggers without exclusivity rule. The multiple insert and delete triggers feature is new in Cheetah.

Access to OLD and NEW row values:

Each trigger can create statements to execute at three events:

- before the statement is executed – BEFORE trigger action
- after the statement is executed – AFTER trigger action
- for each row affected by the statement (after the row has been affected)
  -- FOR EACH ROW (FER) trigger action

The FER triggered-action clause can access applicable OLD and NEW version values for the row. DELETE, UPDATE and SELECT statements have OLD rows, while INSERT and UPDATE statements have NEW rows. UPDATE and INSERT triggers can determine the eventual value inserted or updated using the EXECUTE [PROCEDURE|FUNCTION] … INTO column-name feature of IDS.

INSTEAD OF triggers are created on views for INSERT, UPDATE, or DELETE operations. For updateable views, INSTEAD OF triggers are executed on DML operations on INSTEAD OF triggers on base tables.

IDS 11.10 simplifies the access to the OLD and NEW trigger-correlated variables (values in the affected row) in the SPL procedures invoked by FER trigger-action clauses. After you declare that the procedure is attached to a table, the statements in the procedure can directly access trigger-correlated variables and modify appropriate values through LET assignment statements. These procedures have all the functionality of a normal procedure.

```
create table tab1 (col1 int,col2 int);
create table tab2 (col1 int);
create table temptab1 (old_col1 int, new_col1 int, old_col2 int, new_col2
int);


/*
 This procedure is invoked from the INSERT trigger in this example.

 This function also illustrates 4 new functions in Cheetah:
INSERTING will return true if the procedure is called from the For Each
Row action of the INSERT trigger. This procedure can also be called from
other trigger action statements: UPDATE, SELECT, DELETE.  UPDATING,
SELECTING and DELETING will be true when the procedure is invoked from
the respective trigger action.

*/
create procedure proc1()
referencing OLD as o NEW as n for tab1; -- new syntax.

if (INSERTING) then  -- INSERTING new boolean function
        n.col1 = n.col1 + 1;    -- You can modify the new values.
        insert into temptab1 values(0,n.col1,1,n.col2);
end if

if (UPDATING) then  -- UPDATING new boolean function
        insert into temptab1 values(o.col1,n.col1,o.col2,n.col2);
end if

if (SELECTING) then  -- SELECTING new boolean function
             -- you can access relevant old and new values.
        insert into temptab1 values (o.col1,0,o.col2,0);
end if
if (DELETING) then   -- DELETING new boolean function
        delete from temptab1 where temptab1.col1 = o.col1;
end if
```

```
end procedure;

create procedure proc2()
referencing OLD as o NEW as n for tab2
returning int;

LET n.col1 = n.col1 * 1.1 ; -- increment the inserted value 10%

end procedure;

create trigger ins_trig_tab1 INSERT on tab1 referencing new as post
for each row(execute procedure proc1() with trigger references);

create trigger ins_trig_tab2 INSERT on tab2 referencing new as n
for each row (execute procedure proc2() with trigger references);


insert into tab1 values (111,222);
```

The above statement will execute ins_trigger trig_tab1 and therefore will execute procedure proc1(). The procedure will increment the value of col1 by 1. So, the value inserted will be (112, 222).

```
insert into tab2 values (100);
```

The above statement will execute ins_trigger trig_tab2 and therefore will execute procedure proc2(). The procedure will increment the value of col1 by 10%. So, the value inserted into tab2 will be 110.

Index Self-Join access method

Traditionally an index scan allows one to scan a single range (based on the start/stop key) of an index. The Index Self Join access method lets you scan many mini-ranges instead of a large single range, based on filters on non-leading keys of an index.

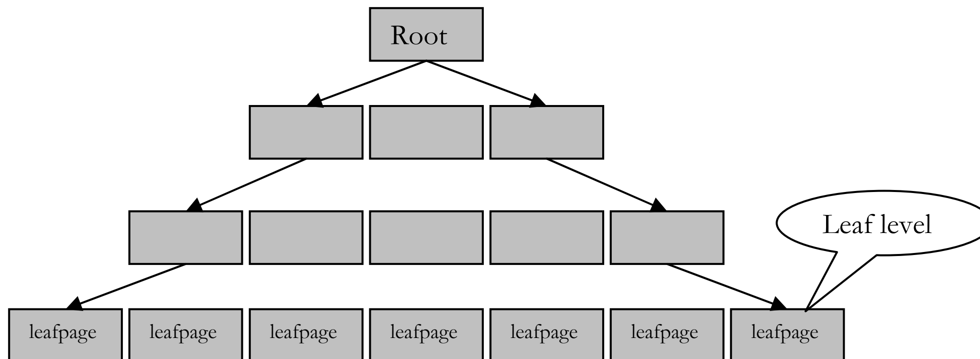
Index self-join is a new type of index scan where the table is logically joined to itself, such that for each unique combination of the leading key column(s) of an index, additional filters on non-leading key columns of the index are used to perform a more efficient mini-index scan. Results from multiple mini-index scans are then combined to generate the result set of the query.

Diagram showing differences in index scans

Here is a before and after illustration of how a sample query will be handled.

```
SELECT * FROM tab
WHERE c1 >= 1 and c1 <= 3 and c2 >= 10 and c2 <= 11 and c3 >= 100 and c3 <= 102
```

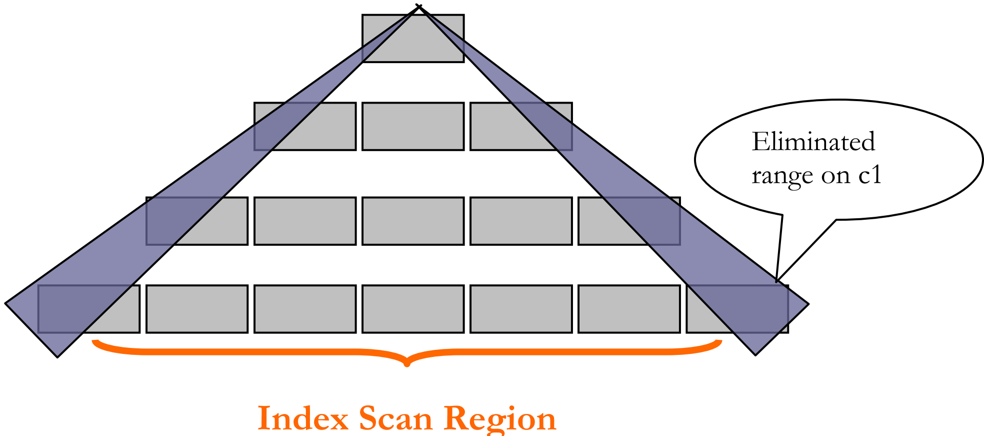
View of the index on (c1, c2, c3)



In prior releases, we could only use filters on c1 (c1 >= 1 and c1 <= 3) for positioning of the index scan:

Prior releases

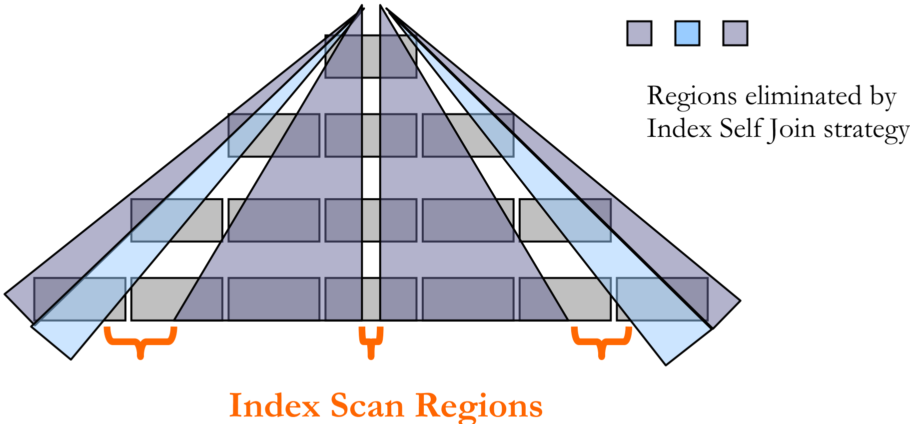
Lower Filter c1 >= 1 Upper Filter c1 <= 3



With this feature, we can use filters on c2 and c3 for positioning of the index scan, which allows us to skip unnecessary index keys at the two ends of the index, and IDS will only scan pockets of the index that are relevant, thereby avoiding scanning a large portion of the index. This strategy will improve the query performance by reducing the portion of the index IDS has to scan.

With this Feature

Lead Keys: c1, c2
Lower Filter c1 = c1 and c2 = c2 and c3 >= 100
Upper Filter c3 <= 102



Optimizer Directives in ANSI-Compliant Joined Queries

Optimizer directives influence the optimizer to choose better access paths, join orders,., for the query execution plan than a plan based only on the available data distribution. Directives help DBAs and developers gain better control over the query plan. IDS 11.10 allows common directives for ANSIJOIN queries. The optimizer attempts to use these directives in appropriate cases. When it cannot, either because an access method is invalid in certain scenarios or cannot be used and still return correct results, the optimizer prints out the directives not used in the explain file.

Previously, ANSI join queries allowed the EXPLAIN, AVOID_EXECUTE and FIRST_ROWS/ALL_ROWS directives. In this release, other directives, such as table access methods (FULL, INDEX, etc.), join methods (USE_HASH, USE_NL, etc), and join order (ORDERED) are allowed in an ANSI join query.

For example: (--+, /*+, {+ are the escape sequences for directives)

Example1: Using directives

```
select --+ INDEX(mytab, myidx)
a.col1, b.coly
from tab1 a, tab2 b
where a.col1 = 1234 and a.col1 = b.colx;
```

Example1: Reading directives in Explain file

```
select --+ FULL(t2), INDEX(t1,t1i1), ORDERED
* from t1 left outer join t2 on (t1.c1=t2.c1 and t1.c1=2)
where t2.c1 is not null

DIRECTIVES FOLLOWED:
FULL ( t2 )
INDEX ( t1 t1i1 )
ORDERED
DIRECTIVES NOT FOLLOWED:

Estimated Cost: 7
Estimated # of Rows Returned: 1

  1) sqlqa.t1: INDEX PATH

        Filters: sqlqa.t1.c1 = 2

    (1) Index Keys: c2   (Serial, fragments: ALL)

  2) sqlqa.t2: SEQUENTIAL SCAN

        Filters:
        Table Scan Filters: sqlqa.t2.c1 = 2


DYNAMIC HASH JOIN
    Dynamic Hash Filters: sqlqa.t1.c1 = sqlqa.t2.c1
```

Improved Statistics Collection and Query Explain File.

The time when a user ran UPDATE STATISTICS LOW on a table is now stored in the systables system catalog table in the ustlowts column (“informix”.SYSTABLES.ustlowts), which has a data type of datetime year to fraction(5). For example:

```
> select a.tabname, a.ustlowts from systables a where tabname = 't1';

tabname          t1
ustlowts         2007-02-05 18:16:29.00000

1 row(s) retrieved.
```

In IDS 11.10, the CREATE INDEX operation automatically updates the statistics about the table and creates distributions for the leading key of the index. Applications no longer need to run UPDATE STATISTICS LOW on the table for the optimizer to start considering the newly-created index; it’s now done automatically. For the B-TREE indexes on basic IDS types, IDS exploits the sort done during index creation to create the distribution for the leading column of the key. For example, when you create a composite index using the key (c1, c2, c3), IDS will automatically create distribution on c1 – no distribution will be automatically created for c2 and c3. IDS uses an innovative way to implicitly create this distribution for fragmented indexes as well. Because creation of distributions requires additional memory, make sure to tune the memory parameters used in update statistics execution. See the developerWorks article “Tuning UPDATE STATISTICS” at: http://www-128.ibm.com/developerworks/db2/zones/informix/library/techarticle/miller/0203miller.html .

For example:

```
create table t1(a int, b varchar(32));
-- load data
set explain on;
create index index_t1_ab on t1(a,b);
```

Contents of sqexplain.out:

```
CREATE INDEX:
=============

Index:          index_t1_ab on keshav.t1
STATISTICS CREATED AUTOMATICALLY:
Column Distribution for:                keshav.t1.a
Mode:           MEDIUM
Number of Bins:       2 Bin size:    38.0
Sort data:          0.0 MB
Completed building distribution in:     0 minutes 0 seconds
```

“informix”.SYSDISTRIB has four new columns:

Column name	Column Type	Comments
smplsize	float	Sampling size specified by the user; 0.0 when unspecified. A value between 0.0 and 1.0 is the *sampling percentage* of rows and a value above 1 is the *number of rows to be sampled.*

rowssmpld	float	Actual number of rows sampled to create the distribution
constr_time	datetime year to fraction(5)	Time when the distribution was created.
ustnrows	float	Number of rows in the table when the distribution was created

Explain File Improvements

You can now set the location of the explain file in the SET EXPLAIN statement; it overrides the default location and file name. For example:

SET EXPLAIN FILE TO "/tmp/cheetah/myexplainfile.out';

When the query is executed, the explain file will include the estimated and actual cardinalities for each table in the query or subquery. For example:

```
QUERY:
------
select a from tab1 where a in (select x from tab2 where tab2.x <
10)

Estimated Cost: 46
Estimated # of Rows Returned: 677

  1) keshav.tab1: INDEX PATH

    (1) Index Keys: a b   (Key-Only)  (Serial, fragments: ALL)
        Lower Index Filter: keshav.tab1.a = ANY <subquery>

    Subquery:
    ---------
    Estimated Cost: 20
    Estimated # of Rows Returned: 87

      1) keshav.tab2: SEQUENTIAL SCAN

            Filters: keshav.tab2.x < 10

Query statistics:
-----------------
  Table map :
  ----------------------------
  Internal name     Table name
  ----------------------------
  t1                tab1

  type     table  rows_prod  est_rows  rows_scan  time       est_cost
  -------------------------------------------------------------------
  scan     t1     584        677       584        00:00:00   47

Subquery statistics:
--------------------

  Table map :
  ----------------------------
```

```
Internal name      Table name
----------------------------
t1                 tab2

type      table  rows_prod  est_rows  rows_scan  time        est_cost
--------------------------------------------------------------------
scan      t1     87         87        500        00:00:00    20

type      rows_sort  est_rows  rows_cons  time
-------------------------------------------------
sort      9          87        87         00:00:00
```

You can enable or disable printing of these statistics by resetting the value of the EXPLAIN_STAT configuration parameter with the onmode command:

```
onmode –wm "EXPLAIN_STAT=1"
onmode –wm "EXPLAIN_STAT=0"
```

The onmode -wm command changes the preference in memory only; this setting is lost when the server is restarted. The onmode -wf command changes the preference in both memory and in the onconfig file.

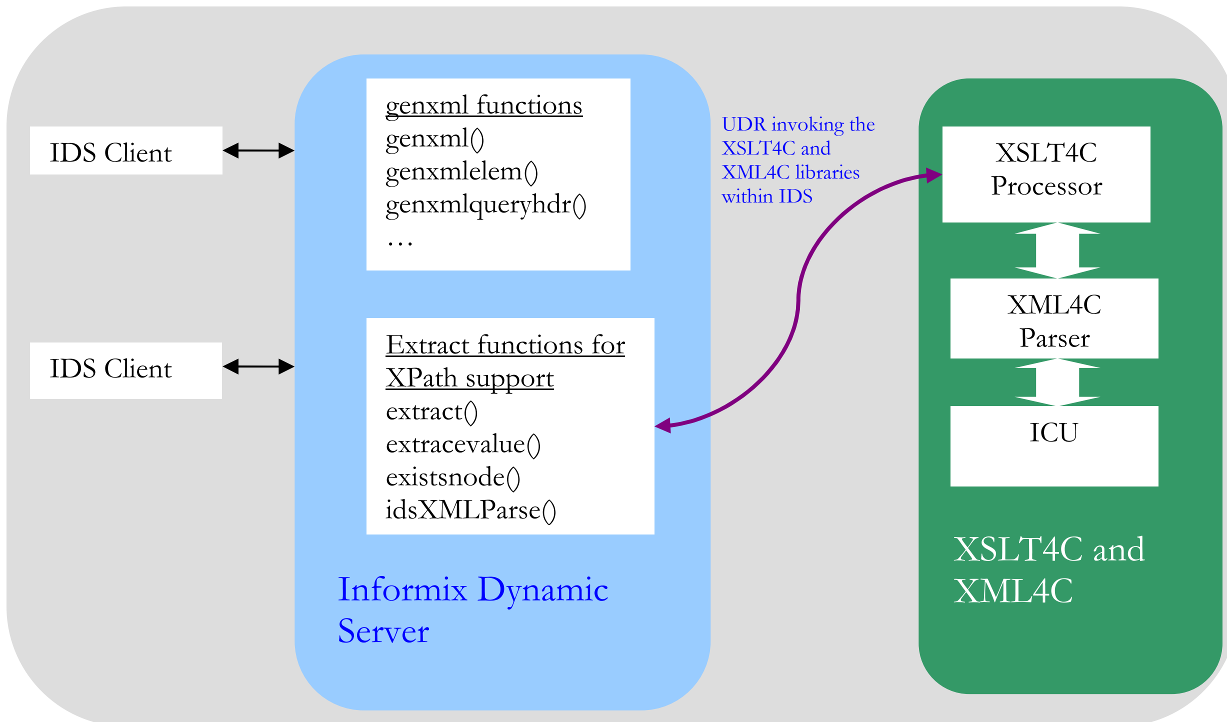
The onmode –Y command enables you to dynamically print query plan on a session:

```
onmode -Y <ses_id> 0       -- off no query plan will be printed.
                   1       -- plan + statistics
                   2       -- plan only
```

XML Publishing and XPath functions

XML is used in document-centric and data-centric applications for integrating heterogeneous data and applications. For example, news feed with pictures, video, text, and data needs a document-centric approach, while an application dealing with stock data or point of sale transactions needs a data-centric approach.

In IDS v11.10, you can use built-in functions to publish IDS result sets as XML documents, to verify that an XML document is well formed, and to extract parts of the document matching an XPATH pattern. These functions can be used directly from SQL statements or in stored procedures. The following illustration shows these functions.



XML Components:

XML4C processor is the open source XERCES XML parser library from IBM. It parses XML documents and can create DOM object for a given XML document. This is used by higher level libraries like XSLT4C to validate and parse XML documents.

XSLT4C processor: Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. XPath is included in the XSLT definition. We use the XPath support to provide XPath functionality in Extract functions.

ICU is the International Component for Unicode used for encoding, formatting and sorting by XML4C and XSLT4C processors.

IDS XML Functionality:

The genxml functions in the example below illustrate publishing a result set into an XML document, and the extract functions illustrate extracting parts of the XML document.

```
EXECUTE FUNCTION genxmlqueryhdr('manufact_set','SELECT * FROM
manufact');

(expression)  <?xml version="1.0" encoding="ISO-8859-1" ?>
              <!DOCTYPE manufact_set SYSTEM
"../manufact_set.dtd">
              <?xml-stylesheet type="text/xsl"
href="../manufact_set.xsl" ?>
              <manufact_set>
              <row>
              <manu_code>SMT</manu_code>
         ……    (removed for brevity)
              <manu_code>NKL</manu_code>
              <manu_name>Nikolus        </manu_name>
              <lead_time>   8</lead_time>
              </row>
              <row>
              <manu_code>PRC</manu_code>
              <manu_name>ProCycle       </manu_name>
              <lead_time>   9</lead_time>
              </row>
              </manufact_set>

1 row(s) retrieved.
```

```
SELECT genxml( customer, 'customer') from customer;
SELECT genxmlelem(manufact, 'manufact') from manufact;

SELECT genxml( ROW(customer_num, fname, lname), 'subset') FROM customer;

SELECT genxml(
ROW(A.customer_num, fname, lname, call_dtime, call_code, call_descr,
res_dtime, res_dtime, res_descr), 'customer_call
FROM customer a, cust_calls b
WHERE a.customer_num = b.customer_num and a.customer_num = 106;

SELECT genxml(ROW(customer_num, fname, lname), 'subset')
FROM customer;
```

The extract functions take an XML document and XPath expression as parameters and extract the match XML node, XML value or verify if the document contains a specified XML pattern.

```
SELECT extract(col2, '/personnel/person[3]/name/given') FROM tab;
SELECT extractvalue(col2, '/personnel/person[3]/name/given') FROM tab;

execute function
extract("<person><name><fname>john</fname><lname>Kelly</lname></person>",
               "/person/name/lname");
```

```
select col1 from tab where existsnode(col2, '/personnel/person/*/email') =
1;

select idsxmlparse(genxmlquery('customer', 'select * from customer where
customer_num = 114 ')) from customer  where customer_num = 114;

SELECT extractclob(col2, '/personnel/person[3]/name/given') FROM
tab_clob_neg;

SELECT extractvalueclob(col2, '/personnel/person[3]/name/given') FROM
tab_clob_neg;
```

Please refer to IDS Cheetah documentation for the complete list of publishing and extract functions provided in IDS 11.10. See the Cheetah Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v111/index.jsp

Acknowledgements

Thanks to IDS development engineers, who developed these features and provided valuable input to this document. Special thanks to Frederick Ho, Senior Product Manager, IBM Informix and Judy Burkhart, IBM Information Development Manager for suggestions and corrections -- both improved the quality of the document.