Db2 for Linux, Unix, and Windows

Version 11+ Highlights

Actionable Insights
Continuous Availability
Massive Scalability
Outthink the Possible

George Baklarz and Enzo Cialini

Db2 for Linux, Unix, and Windows: Version 11 Highlights

Initial Publication: October 23, 2016
Updated: May 16, 2017
Updated: June 20, 2017
Updated: October 2, 2017

Revisions

Initial Publication: October 23, 2016

Db2 11.1.1.1: May 16, 2017

Some Fix pack 1 (Mod 1) features were added to the book. Changes in the book are marked with the number ❶ at the beginning of the section. Corrections were also made to some text and examples but are not marked.

Db2 11.1.2.2: June 20, 2017

Some Fix pack 2 (Mod 2) features were added to the book. Changes in the book marked with the number ❷ at the beginning of the section. This version of the eBook also introduces the new spelling of Db2. It is not a spelling mistake!

Db2 11.1.2.2: October 2, 2017

Some additional information added on new editions, packaging, and features in the latest fix pack.

About the Authors

George Baklarz, B. Math, M. Sc., Ph.D. Eng., has spent 31 years at IBM working on various aspects of database technology. George was part of the team that helped move the OS/2 ES database to Toronto to become part of the Db2 family of products. He has worked on vendor enablement, competitive analysis, product marketing, product planning, and technical sales support. George has written 10 books on Db2 and other database technologies. George is currently part of the Worldwide Core Database Technical Sales Team.

Enzo Cialini, B.Sc., is a Senior Technical Staff Member and Master Inventor in the Worldwide Core Database Technical Sales Team and formerly the Chief Quality Assurance Architect for Db2 & PureData in the IBM Toronto Db2 Development Team. He is also a published book author and written various papers on Db2. Enzo has 25 years of experience in database technology, software development, testing, support, competitive analysis and production deployments.

Forward

The new economy is changing the way we gather information, manage data, gain insights, reinvent our businesses, and do so quickly and iteratively. The digital transformation of everything is changing the information value chain not just from your own organization, but from stakeholders and third-party data providers. And with this new information and in context from which they're gathered, we gain a deeper understanding of client behaviors and market conditions, and formulate faster responses to changing competitive landscape.

The advent of cloud and mobile computing further encourages the distribution and consumption of information at a rapid pace, which in turn generates more information to analyze.

Therefore, this new economy requires a robust database software that sits at the heart of an enterprise to support its transformation in this new era of digital, cloud and cognitive.

Db2 is a versatile and scalable database that can support your transactional and analytical workloads for different applications whether on premises or in the cloud up to petabytes in volume. It is easy to deploy, upgrade and maintain and at the same time it simplifies the move from Oracle databases by leveraging the SQL Compatibility component along with your existing skills and assets with minimal changes to the application.

Db2's pureScale capability ensures that your business data is always available and accessible without any interruptions to your business processes, ensuring that your consumers are not impacted wherever they may be.

With BLU MPP, you leverage multi parallel processing with in-memory columnar technology to provide extreme performance enhancements for your analytic workloads.
Finally, Db2 provides the flexibility of deploying on premise or on the cloud with integrated security to ensure that your new business process applications and data are where they make sense, near your consumers.

Several other distributed database offerings are available from IBM in a managed environment, hosted environment, virtualized, using your own data center, or on an appliance:

- Db2 Warehouse (Local Docker Container)
- Db2 Warehouse on Cloud (Managed Warehouse Service)
- Db2 on Cloud (Managed OLTP Service)
- Db2 Hosted
- Informix
- Informix on Cloud
- PureData System for Analytics
- IBM Integrated Analytics System
- Cloudant
- IBM Db2 Event Store

No matter what your requirements are, IBM has a database offering that will meet your needs.

Acknowledgments

We want to thank the following people who, in one way or another, contributed to this book:

- Michael Springgay, Kelly Schlamb, Paul Bird, Matt Huras, Calisto Zuzarte, John Hornibrook, Greg Stager, and Alan Lee

We would also like to thank all the development team for helping to deliver this release given the tremendous deadlines and constraints that they have been under.

The contents of this eBook are the result of a lot of research and testing based on the contents of our Db2 11.1 Knowledge Center. The authors of our online documentation deserve special thanks for getting the details to us early in the product development cycle so we could build much of our material.

For the most up-to-date information on Db2 11.1 features, please refer to the IBM Knowledge Center:

http://www.ibm.com/support/knowledgecenter/SSEPGG

There you can select the Version 11.1.0 release from the drop-down menu and get more details on the features we explore in this eBook.

Introduction

About This Book

The Db2 11.1 release delivers several significant enhancements including Database Partitioning Feature (DPF) for BLU columnar technology, improved pureScale performance and further High Availability Disaster Recovery (HADR) support, and numerous
SQL features.

This book was written to highlight many of the new features and functions that are now available in this release, without you having to search through various forums, blogs, and online manuals. We hope that this book gives you more insight into what you can now accomplish with Db2 11.1, and include it on your shortlist of databases to deploy, whether it is on premise, in the cloud, or a hybrid approach.

George and Enzo

How This Book Is Organized

We organized this book into 11 chapters that cover many of the highlights and key features found in the Db2 11.1 release.

- Chapter 1 gives you an overview of the new packaging changes in Db2 11.1, along with pre-requisites for installing the product
- Chapter 2 discusses the enhancements to Db2 BLU, including support for BLU columnar tables in a DPF (Database Partitioning Feature) environment
- Chapter 3 examines what pureScale capabilities provide you in a production environment, along with all the enhancements that have been made to simplify installation and management of a cluster
- Chapter 4 through 12 are focused on SQL and compatibility enhancements

1

Packaging, Installation, and Administration

ENHANCED LICENSING AND DEPLOYMENT OPTIONS FOR DB2

Packaging, Installation, and Administration

Db2 11.1 introduced several pricing and packaging changes that will affect some customers. In addition, the end of service of Db2 9.7 and Db2 10.1 were announced, which means that customers with either of those releases will need to start making plans to upgrade to a new version of Db2 over the next year.

End of Service and Marketing

Three additional announcements were made regarding the end of marketing and the end of service for older Db2 releases and features. Db2 10.5 end of marketing, on September 30th, 2016

- Db2 9.7 and Db2 10.1 end of service, on September 30th, 2017
- Discontinued Business Value Offerings (BVO)

End of marketing means that customers will not be able to purchase Db2 10.5 after the end of September 2016. This means that you can only purchase Db2 11.1 now, but you are still able to get copies of Db2 10.5 if required. Customers continue to get service and support for Db2 10.5 since end of service is not being announced now.

For those customers running either Db2 9.7 or Db2 10.1, they will need to begin planning on upgrading to either Db2 10.5 or Db2 11.1. Extended service contracts will be available to maintain your old database images after September 2017, but there will be an additional charge to do that.

For those customers who purchased any of the BLU Acceleration Business Value Offerings, they will be upgraded to an Advanced license of Db2 depending on which license they had when they purchased the BVO. The Encryption Offering and the Business Application Continuity Offering are bundled with every edition of Db2 11.1, so customers that purchased that feature will no longer have to pay service and support on it when upgrading to Db2 11.1.

New Included Features

One significant change to Db2 packaging is that all Db2 Editions (Workgroup, Enterprise, Advanced Workgroup and Advanced Enterprise, Digital Standard, Digital Advanced) will include Encryption, Federation, and pureScale. While there are limitations on the use of Federation and pureScale in Workgroup, Enterprise, and Digital Standard Editions, the Encryption feature is fully supported across all editions.

Federation

Db2 11.1 now includes data federation and virtualization through a robust SQL interface and a full relational database engine designed for global optimization of data access. This capability offers homogeneous or heterogeneous federation (depending on the Db2 Edition) between select IBM and non-IBM data sources. You can quickly prototype and implement solutions involving disparate data by virtualizing access and optimizing performance. Data can remain in place for both transactional as well as analytical workloads.

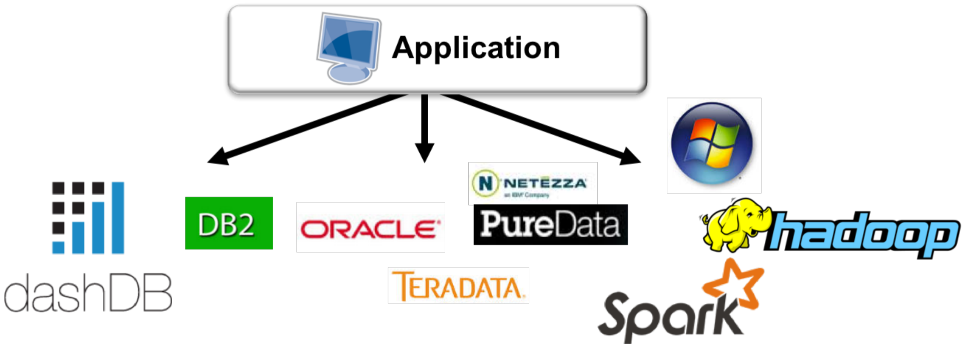


Figure 1: Federation Support

Db2 Advanced Server Editions include all of the wrappers that are part of the federation support. The Db2 Workgroup and Enterprise Edition products only include support for connecting to Db2 and Informix data sources.

Encryption

Db2 included encryption at the database level in Version 10.5. However, the encryption feature required the use of a local keystore to contain the master key for the database. This required that the keystore itself be backed up and managed independently from the database backup.

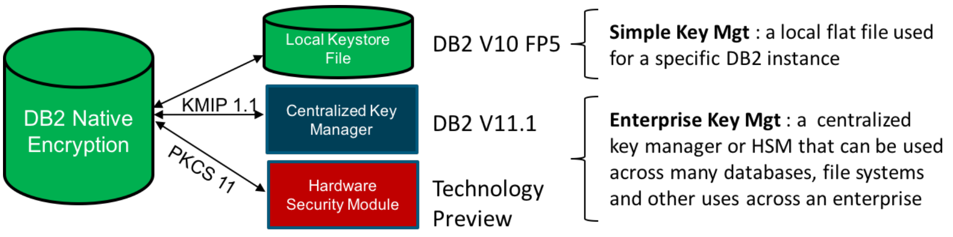

DB2 Native Encryption
KMIP 1.1
PKCS 11
Local Keystore File
Centralized Key Manager
Hardware Security Module
DB2 V10 FP5
DB2 V11.1
Technology Preview
Simple Key Mgt : a local flat file used for a specific DB2 instance
Enterprise Key Mgt : a centralized key manager or HSM that can be used across many databases, file systems and other uses across an enterprise

Figure 2: Encryption Support

Rather than manage several separate keystores manually, customers can now use any KMIP 1.1 centralized key manager to manage the Db2 keys. The ISKLM product (IBM Security Key Lifecycle Manager) has been validated to work with Db2, but any KMIP 1.1 compliant manager is supported.

ISKLM helps meet regulations and standards such as:

- Payment Card Industry Data Security Standard (PCI DSS)
- Sarbanes-Oxley
- Health Insurance Portability and Accountability Act (HIPAA)

Using the PKCS#11 protocol, Db2 Native Encryption will be enhanced to support using Hardware Security Modules (HSM) as the keystore for the master key. The initial set of HSM products validated will include Safenet Luna SA and Thales nShield Connect+, but any PKCS#11 complaint product is supported.

➊HSM support is now available in Db2 11.1.1.1. More detailed information on the supported environments can be found in the fix pack support information on: http://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.kc.doc/welcome.html

pureScale Administration Feature

Db2 11.1 Workgroup, Enterprise, and Digital Standard Edition include the pureScale feature, limited to two member clusters. Customers who do not require the scalability of pureScale but require the benefits of continuous availability, can do so now at lower costs.

The limited two-member pureScale cluster configuration enables one member to process application workloads, and a second member to provide continuous availability when the first member is undergoing planned or unplanned outages. In addition, the second member can also be used to perform administrative tasks and utilities, thus off-loading these tasks from the primary member.

The first member is fully licensed, while the second member only requires minimal licensing similar to an HADR environment (licensed as warm standby, e.g. 100 PVUs or 1 VPC).

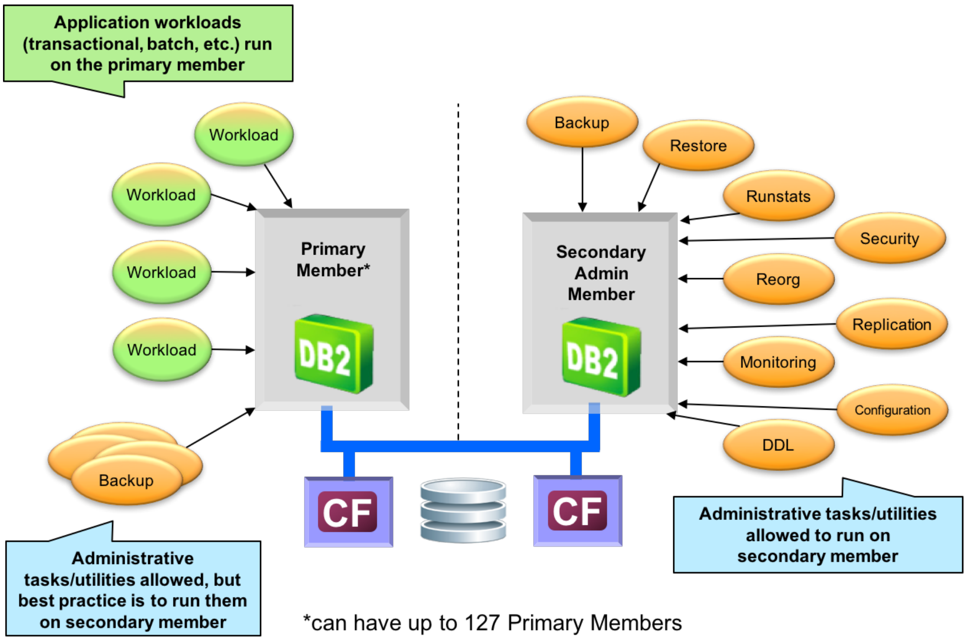

Application workloads (transactional, batch, etc.) run on the primary member
Workload
Workload
Workload
Workload
Backup
Primary Member*
DB2
Secondary Admin Member
DB2
CF
CF
Backup
Restore
Runstats
Security
Reorg
Replication
Monitoring
Configuration
DDL
Administrative tasks/utilities allowed to run on secondary member
Administrative tasks/utilities allowed, but best practice is to run them on secondary member
*can have up to 127 Primary Members

Figure 3: pureScale 2-node configuration

All application workloads are directed to the primary active member(s), sometimes referred to as the "primary" member while utilities and administrative tasks are allowed on the secondary member. This configuration is ideal for off-loading backups from primary members as well as ensuring continuous availability in the event of a member failure.

Administrative tasks and utilities, however, are allowed to run on the secondary member. Nothing is preventing the same administrative work from running on the primary members as well, but the best practice is to keep administrative tasks to the secondary member - thus, freeing the primary member resources for processing the workload.

This is an active/active cluster as work is being done on ALL members of the cluster but using an 'active/passive' licensing model. The same 'active/passive' licensing model is available to be applied to the fully scalable Db2 pureScale active/active configuration available in Db2 Advanced Editions, including the new Direct Advanced Edition, allowing for off-loading of administration tasks and utilities in a continuously available and scalable >2-member cluster.

Db2 Express and Db2 Express Community Editions

For those users looking to develop, deploy and distribute Db2 at NO charge should look at Db2 Express-C. Db2 Express-C Edition is a no-charge community edition of Db2 available for Windows and Linux platforms, which provides the core features of Db2 Editions.

For more advanced features, such as Db2 pureScale and BLU Acceleration, the solution can be upgraded without requiring modification to your applications.

Db2 Express-C offers these features and benefits:

- Encryption for data at rest and in-transit to protect your data
- Uses Time Travel Query to provide bi-temporal data management
- Delivers an integrated tools environment with IBM Data Studio for easier management and development
- Contains the SQL compatibility feature to more easily run applications written for other databases on Db2
- Continuous Data Ingest
- Federation
- pureXML storage
- JSON NoSQL

The licensed Db2 Express Edition is no longer offered in Db2 11.1. Those customers that currently have Db2 Express will be upgraded to the Db2 Workgroup Edition.

Db2 Workgroup and Enterprise Editions

Both Db2 Workgroup and Enterprise Editions have had several additional features included into the packaging:

- pureScale Standby Member Option
- Table partitioning, Encryption
- Federation (Db2 & Informix)

In addition, Db2 Workgroup Edition has had further features added which are already included in Db2 Enterprise Edition

- Multi-dimensional Clustering
- Multi-Temp Storage
- Query Parallelism

There are some exclusions to what is included in the Workgroup and Enterprise editions. The list includes (but it not limited to):

- Data Partitioning
- SQL Warehouse (SQW)
- Materialize Query Tables (included in Enterprise Edition)
- BLU Acceleration, Compression

The one big change in the licensing of Db2 Warehouse Edition is that there is no longer a cap on the size of database. The core limit remains the same at 16 cores, but the amount of main memory has increased to 128GB. There are no limits on the cores, memory or database size with Db2 Enterprise Edition.

There are two optional products that customers may want to purchase for Workgroup and Enterprise Edition:

- Db2 Performance Management Offering (WLM) which is part of the Data Server Manager Enterprise Edition
- Advanced Recovery Feature

These products are described in another section.

Advanced Workgroup and Advanced Enterprise Editions

As with the prior release of Db2 10.5, the Advanced Editions of Workgroup and Enterprise Edition include all features for one single price. What is new in Db2 11.1 is the new Federated feature which includes connectivity to Db2, Informix, and many other relational and unstructured data sources.

There are three products that are included as part of the Advanced Editions that have limitation on usage. These products are:

- Db2 Connect included for using SQW tooling to access Db2 for z and Db2 for i
- InfoSphere Data Architect limited to 10 authorized users
- Cognos Analytics limited to 5 authorized users

The only optional feature that customers may want to consider is the Advanced Recovery Feature.

Additional Products for Db2

In conjunction with the Db2 11.1 database release, several tools were released that are included in the Advanced Editions of Db2, and available for purchase with the other editions.

Advanced Recovery Feature

This feature consists of advanced database backup, recovery, and data extraction tools that can help you improve data availability, mitigate risk, and accelerate crucial administrative tasks when time is of the essence.

New features of the tools include support for Db2 11.1, including using the tools with Db2 Native Encryption.

The Advanced Recovery Feature bundle contains the following new releases of the included tools:

- Db2 Merge Backup for Linux, UNIX, and Windows V3.1
- Db2 Recovery Expert for Linux, UNIX, and Windows V5.1
- Optim High Performance Unload for Db2 for Linux, UNIX, and Windows V6.1

Db2 Performance Management Offering

The Db2 Performance Management Offering (WLM) is part of the Data Server Manager Enterprise Edition. Db2 Performance Management Offering helps organizations manage their Db2 databases across the enterprise through tooling that has a common integrated console, enhanced simplicity, improved performance and scalability, and simplified workflow with actionable recommendations. Organizations can do the following:

- Simplify database administration and reduce errors with integrated web console to view and monitor the health of databases across the enterprise.
- Identify, diagnose, prevent, and solve performance problems with a more predictable database server execution environment.
- Track and manage client/server and database environment efficiently with centralized management.
- Optimize database performance with expert recommendations on query design, unique indexes, and statistics quality.

Db2 Direct Editions

Db2 11.1 includes a new delivery mechanism for licensing. The Db2 Direct Editions are intended to make it easier to acquire Db2 licenses for cloud deployments:

- New license metrics to facilitate hybrid cloud deployments
- Acquire the product directly online (Passport Advantage)
- Option to deploy either on-premises or on cloud

It doesn't matter where you place your Db2 server, on premise, on cloud, etc. You are licensed by the number of Virtual Processor core vCPUs (explained on the next page). The license is totally flexible and can be moved between platforms without any change in license fees. Note that this license is an OPEX (Operating Expense) license – you are renting or leasing the Db2 license. The price includes the use of the product as well as service and support. If the license is not renewed, the database will not have a valid license.

There are two Db2 Direct Editions, depending on what the database requirements are:

- Db2 Direct Standard Edition 11.1 - Has all the database features of Db2 Workgroup Server Edition
- Db2 Direct Advanced Edition 11.1 - Has all the database features of Db2 Advanced Enterprise Server Edition

Both licenses are sold as a monthly license charge using the Virtual Processor Core (VPC) sold as a monthly license charge. Customers can choose to maintain these databases with either:

- Long Term Support Release (LTSR) support
- Continuous Delivery (CD) support

LTSR support will give the customer fix packs only with no additional functionality added to the product. CD support will contain the fixes found in the LTSR stream and may also contain new functionality and enhancements.

❷Db2 Developer-C (Community) Edition

Db2 11.1.2.2 introduces a new version of the Db2 Developer Edition. This new Db2 Developer-C Edition has similar characteristics to the existing Developer Edition, except that it is free to download and use by all users. The product is not supported (similar to Express-C and Trial editions) and cannot be used for production. It has no limits on database engine functionality but it does have limits on the environment (similar to Express-C).

Why is this new edition important and why was it introduced? Being a free download and without functional limitations, developers can download and use this for development purposes. Companies can use this for a start-up project without cost. It does not expire, and thus can be used for a long period of time (non-production only). Being full function, developers and teams can test out the many advanced features they may not be entitled to in their other for fee systems, such as compression, MPP, BLU Acceleration, pureScale, etc. Savings can be had in non-production environments where support is not required (when support is required – a for fee edition must be used).

There are a few limitations when using Developer-C. The maximum machines size when using this product is:

- 4 cores, 16GB of memory
- 100GB of data in user tablespaces

Some of the supporting programs are not included: Cognos, IDA, WebSphere AS / MQ, and DSM Enterprise Edition.

❷Db2 Developer Community Edition

A new Download and Go environment has been developed that includes a fully configured and functional version of Db2 Developer-C in a Docker environment. This new Download and Go approach will make getting a copy of Db2 up and running a much easier experience. From 10+ screens and multiple inputs, the new approach requires only three clicks and a userid and password to get started. The necessary Docker components are downloaded and configured for you or an existing Docker installation can be used.

The download is relatively small (about 15 minutes) and executable files are available for Mac, Windows, and Linux environments. The download package includes:

- Db2 Developer-C 11.1.2.2
- Data Server Manager 2.1.4
- Data Studio (optional)

This package has the same limitations (non-warranted) as Developer-C Development such as non-production use only and core, memory, and storage limitations.

Operating System Support

A customer needs to be aware that there are new operating systems (O/S) requirements for Db2 11.1. While their existing Db2 installation may be supported at an older O/S release level, they will not be able to upgrade their database unless the O/S has been updated as well.

Db2 11.1 introduces support for Power Linux LE (Little Endian) which also includes support for pureScale. Db2 supports Power Linux LE with the following three operating systems:

- Red Hat Enterprise Linux (RHEL) 7.1+
- SUSE Linux Enterprise Server (SLES) 12
- Ubuntu 14.04 LTS

Db2 11.1 has also updated the operating systems that it supports. Note the new requirements in the following list. Support for 32-bit server operating systems has been eliminated for new instances in this release, so Windows 32-bit and Linux 32-bit are no longer supported for the server editions but continue to be available as part of the developer edition. 32-bit clients continue to be supported however.

- Intel 64-bit
    - Windows 7, 8.1, 10, Windows Server 2012 R2
    - Red Hat Enterprise Linux (RHEL) 6.7+, 7.1+
    - SUSE Linux Enterprise Server (SLES) 11SP4+, 12
    - Ubuntu 14.04 LTS
- AIX Version 7.1 TL 3 SP5+
- zLinux
    - Red Hat Enterprise Linux (RHEL) 7.1+
    - SUSE Linux Enterprise Server (SLES) 12

Starting in Db2 11.1, some operating systems, Linux distributions and architectures are no longer supported. Customers need to look at alternative platforms to upgrade to Db2 11.1.

The following are no longer being supported in Db2 11.1.

- HP-UX, Solaris
- Power Linux BE, Inspur K-UX

However, customers who are currently on Db2 10.5 will continue to receive service for these platforms until the End of Service date for Db2 10.5.

Virtualization Support

Db2 11.1 supports a number of virtualization environments. Virtualization can improve the utilization of processor/machine resources and also give the customer an environment to test systems without having to invest in entire systems. The following is a list of virtualized environments that Db2 supports:

- IBM System z
    - IBM Processor Resource/System Manager
    - z/VM and z/KVM on IBM System z
- IBM Power
    - IBM PowerVM and PowerKVM and IBM Workload Partitions on IBM Power Systems
- Linux X86-64 Platforms
    - Red Hat KVM

    - SUSE KVM
- VMWare ESXi
- Docker container support – Linux only
- Microsoft
    - Hyper-V
    - Microsoft Windows Azure on x86-64 Windows Platforms only
- pureScale support on Power VM/KVM, VMWare, and KVM

You will note that pureScale is supported in a virtualized environment and that ROCE Adapters can now be virtualized in a VMWare environment (initially) so that multiple members can share the same hardware adapter.

Upgrade Support

Upgrading to Db2 11.1 is supported from Db2 10.5, Db2 10.1, or Db2 9.7. If you have an earlier version of Db2, you must first upgrade to Db2 10.5, Db2 10.1, or Db2 9.7 before upgrading to Db2 11.1.

- Upgrading to a Db2 11.1 non-root installation is supported from a Db2 10.5, Db2 10.1, or Db2 9.7 non-root installation. Upgrading to a Db2 11.1 non-root installation from a pre-Db2 11.1 root installation is not supported.
- Instance bit size is determined by the operating system where Db2 11.1 is installed, and support for 32-bit kernels and 64-bit kernels has changed.
- Upgrading from a system with multiple copies of Db2 10.5, Db2 10.1, or Db2 9.7 is supported. On Windows operating systems, you must be aware of the restrictions on coexistence of previous versions of the Db2 database products.
- Upgrading from a partitioned database environment with multiple database partitions is supported.

Restoring full database offline backups from pre-Db2 11.1 copies is supported. However, rolling forward of logs from a previous level is not possible. Review Backup and restore operations between different operating systems and hardware platforms for complete details about upgrade support using the RESTORE DATABASE command.

Administration Improvements

Db2 11.1 includes a number of administrative enhancements that customers will find useful. Five of them that are highlighted here include:

- Range Partition Table Reorganization
- ADMIN_MOVE_TABLE improvements
- Remote Storage Option
- Db2 History File Backup
- Backup Compression Hardware Acceleration

Range Partition Reorganization

The manageability of large range partitioned tables has been improved in Db2 11.1. A single partition of a range partitioned table can now be reorganized with the INPLACE option if:

- the table has no global index (i.e. non-partitioned indexes)
- ON DATA PARTITION is specified

The reorganization can only occur on one data partition at a time, and the table must be at least three pages in size. The biggest restriction of this feature is that the table cannot have any global indexes. If a global index exists, the reorg command will not run.

ADMIN_MOVE_TABLE

There are two new options in the ADMIN_MOVE_TABLE command:

- REPORT
- TERM

The REPORT option can be used to monitor the progress of table moves. The command calculates a set of values to monitor the progress of single or multiple table moves. The focus is on the COPY and REPLAY phase of a running table move.

The REPORT option requires a table schema and name to get information on a specific table move. If these values are left blank or NULL, information on all outstanding table moves is retrieved.

The TERM option can be used to terminate a table move in progress. TERM will force off the application running the table move, roll back all open transactions and set the table move to a well-defined operational status. From here, the table move can be cancelled or continued.

Remote Storage

Db2 11.1 delivers more flexibility and options for acquiring, sharing and storing data files and backup images, by allows customers to use remote storage for several Db2 utilities:

- INGEST, LOAD, BACKUP, and RESTORE

Db2 supports remote storage using storage aliases for:

- IBM® SoftLayer® Object Storage
- Amazon Simple Storage Service (S3)

Db2 History File Backup

The Db2 history file contains information about log file archive location, log file chain etc. If you use snapshot backups you want to have current information about log file location to perform point in time recovery (RECOVER command). RECOVER needs a current version of the history file to be available.

The NO TABLESPACE backup option allows you to create a current and consistent backup of the history file in a convenient way. A NO TABLESPACE backup does not contain tablespaces, only the history file. A no tablespace backup is used to quickly backup the history file and can be done at different frequency than a database backup. Restore the history file by using the existing HISTORY FILE option with the RESTORE DATABASE command.

Hardware Backup Compression

Db2 backup and log archive compression now support the NX842 hardware accelerator on POWER 7+ and POWER 8 processors. Db2 BACKUPs require the use of a specific NX842 library to take advantage of the hardware acceleration.

```
backup database <dbname> compress comprlib libdb2nx842.a
```

The registry variable DB2_BCKP_COMPRESSION can be set to NX842 for the default BACKUP command to use the hardware acceleration.

Allowing for existing customers that are using the built-in Db2 backup compression (i.e. software compression) to benefit without any changes to scripts/jobs.

Once the registry variable is set, using the following backup command format will benefit from the hardware compression accelerator:

```
backup database <dbname> compress
```

Log archive compression is also supported and can be configured by updating the database configuration parameter LOGARCHCOMPR1 or LOGARCHCOMPR2 to NX842:

```
update database configuration for <dbname>
  using LOGARCHCOMPR1 NX842
```

2

Db2 BLU Enhancements

Extending the Power of In-memory Computing

Db2 BLU Enhancements

There has been significant enhancements and capabilities added to Db2 BLU tables from architecture to performance improvements in the columnar engine. Db2 11.1 adds support for column-organized tables in a database partitioned environment provided by Db2 Database Partition Feature (DPF) allowing for petabyte scaling of BLU tables in a massively parallel processing (MPP) architecture.

In addition to MPP enablement of columnar tables, Db2 11.1 delivers advances in the core BLU Acceleration technology for column-organized tables. These advances include Nested Loop Join (NLJN) support, query rewrite, faster SQL MERGE, enhancements to memory management, further SIMD advances, industry leading parallel sort, improved sortheap utilization, increased SQL parallelism and wide variety of push down functionality into the BLU columnar engine. Other enhancements include BLU Acceleration support for IDENTITY and EXPRESSION generated columns, European Language support and NOT LOGGED INITIALLY support for column-organized tables. Further information on some of these enhancements are covered below.

The improvements not only improve performance of columnar tables, but also increase concurrent intra-parallelism delivering increased queries per hour, as seen in Figure 4, DB Insight query performance.

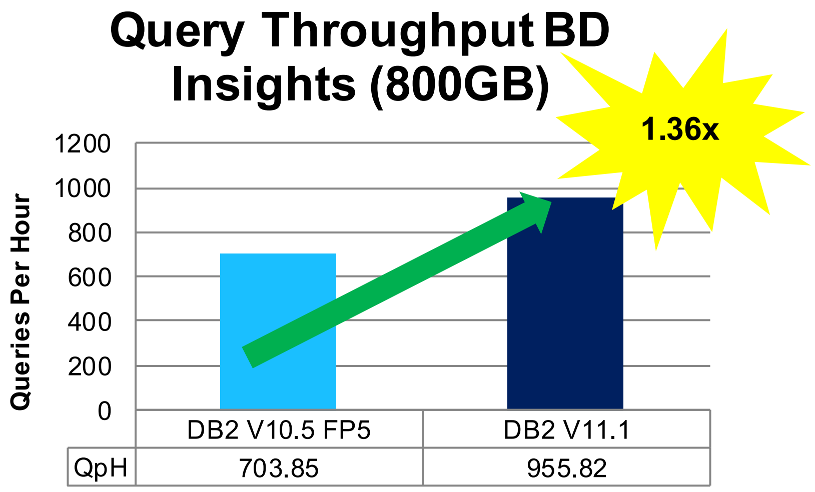


Figure 4: DB Insight Query Performance

Massively Parallel Processing Architecture

With Db2 11.1, you can create column-organized tables in a partitioned database environment with the Database Partitioning Feature of Db2. This allows you to leverage BLU enhancements in a MPP environment. In a DPF shared-nothing database, data is partitioned across multiple servers or partitions with each partition having its own dedicated resources – memory, processors and storage – to process data locally and parallel.

Db2 allows multiple logical partitions to be deployed on the same physical server. Each partition is responsible for a single portion (i.e. partition) of the data in a table and the queries are broken up across all partitions, with partitions working in parallel, leveraging the compute power of all servers for fast query processing. The partitioning of the table is done using a distribution key to distribute table data across a set of database partitions. A distribution key consists of one or more columns of the table. To create a partitioned column-organized table is the same as for row-organized tables, via the DISTRIBUTE BY clause of the CREATE TABLE statement, as below.

```
CREATE TABLE sales(…)
  ORGANIZE BY COLUMN
  DISTRIBUTE BY (C1,C2)
```

If DISTRIBUTE BY HASH is specified, the distribution keys are the keys explicitly included in the column list following the HASH keyword (C1 and C2 in the example above). The columns specified must be a subset of any unique or primary key on the table. Implicit selection of distribution key occurs if the DISTRIBUTE BY HASH clause is omitted and the table is defined in a database partitioning group with multiple partitions or if DISTRIBUTE BY RANDOM clause is used. There are two methods that Db2 uses to distribute data when DISTRIBUTE BY RANDOM is specified:

- Random by unique: If the table includes a unique or primary key, the columns of the unique or primary key are used as the distribution key.

- Random by generation: Db2 will include an IMPLICITLY HIDDEN column in the table to generate and store a generated value to use in the hashing function. The value of the column will be automatically generated as new rows are added to the table. By default, the column name is RANDOM_DISTRIBUTION_KEY. If it collides with the existing column, a non-conflicting name will be generated by Db2.

Fast Communication Manager

The Fast Communication Manager (FCM) communication infrastructure of Db2 DPF has been optimized for columnar data exchange. Thus, data exchange during distributed joins and aggregation processing occurs entirely within the BLU runtime in native columnar format.

The following specific enhancements pertain to column-organized tables in a partitioned environment:

- MPP aware query planning and optimization for the column vector processing engine
- An optimized vector format for columnar data exchange between partitions
- A common table dictionary allowing data to remain compressed across the network
- An optimized communications infrastructure designed for multi-core parallelism

Built on Db2's DPF capabilities means BLU tables are able to scale-out well beyond that of a single server for improved response time, handling larger data sets in the multi-petabyte range and means that existing Db2 MPP data warehouses can easily leverage the in-memory optimized columnar technology.

Faster SQL MERGE processing

The MERGE statement updates a Table or View and uses data from a source (result of a table reference). Rows in the target table that match the source can be deleted or updated as specified and rows that do not exist in the target can be inserted. When the target is a view, updates, deletes or inserts occur to the row in the table(s) on which the view is defined. The following SQL is an example of a MERGE statement.

```
MERGE INTO archive ar
  USING
   (SELECT activity, description FROM activities) ac
  ON (ar.activity = ac.activity)
  WHEN MATCHED THEN
    UPDATE SET description = ac.description
  WHEN NOT MATCHED THEN
    INSERT (activity, description) VALUES
     (ac.activity, ac.description);
```

MERGE has already been supported against columnar-originated tables, however Db2 11.1 delivers performance improvements by pushing the MERGE operation down into the columnar engine which means it operates in native columnar format. This along with a faster sort engine improves performance also.

Nested Loop Join Support

Db2 11.1 delivers Nested Loop Join support for columnar-oriented tables. A Nested Loop Join is one of three techniques that the Db2 Optimizer uses to perform a join between two tables. HASH join is the other join technique that already supports columnar-oriented tables.

A nested-loop join is performed by scanning the inner table for each accessed row of the outer table or by performing an index lookup on the inner table for each accessed row of the outer table. Nested Loop Join support on columnar-oriented tables means that evaluations and activities can be done natively in BLU, thereby improving performance dramatically. There is no external enablement or setting to benefit from this new capability. This performance benefit is realized automatically when the Db2 Optimizer selects a Nested Loop Join as the join technique.

Sort Processing Enhancements

Db2 11.1 includes sort innovations, PARADIS, from the IBM TJ Watson Research division. PARADIS is an efficient parallel algorithm for in-place Radix sort. In addition to delivery industry leading parallel sort, Db2 11.1 is also able to sort compressed and encoded data. This allows more efficient and improved performance as processing is performed within the BLU engine.

Db2 11.1 BLU sort enhancements can increase BLU performance by as much as 13.9x. Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon many factors, including considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve results like those stated here.

More details on PARADIS can be found in the IBM Research Paper: http://www.vldb.org/pvldb/vol8/p1518-cho.pdf

The Db2 Explain facility can be utilized to examine the associated query access plans. The access plans will show what part of the processing is done in the row engine vs the BLU engine. All parts below the "CTQ" evaluator are done in the BLU engine. In Figure 5, we can see that by pushing down the SORT evaluator we keep the data processing in BLU only which contributes to the 4x speedup observed for this query.

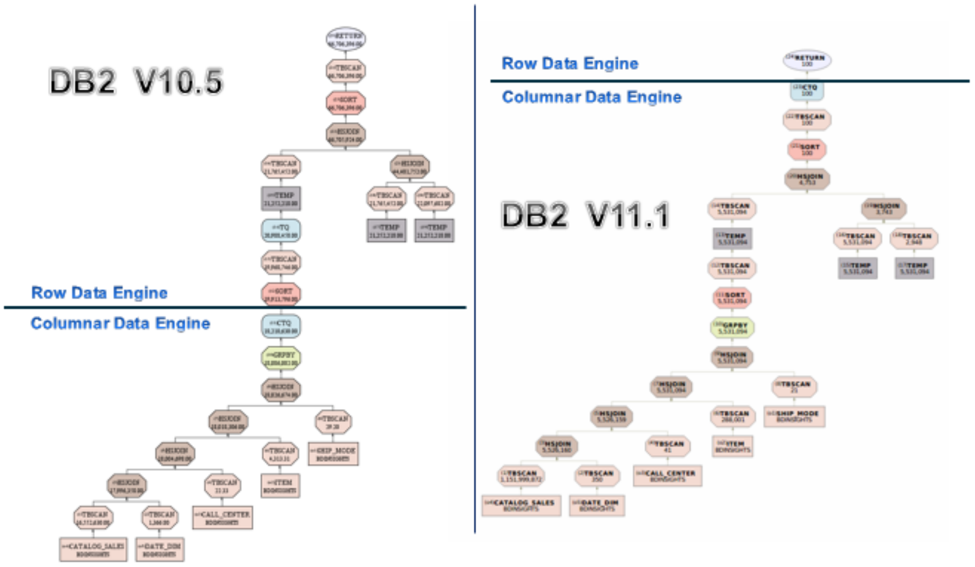
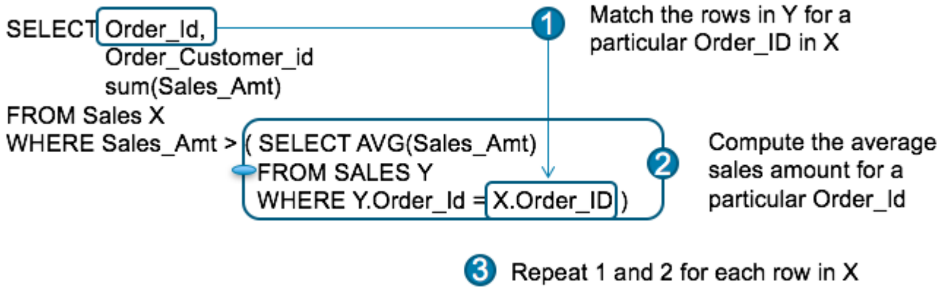


Figure 5: Columnar Engine Native SORT Example

Query Rewrite Enhancements

There have been enhancements to Query Rewrite when dealing with Correlated Subqueries. The enhancement includes the decorrelation of correlated subqueries.

Figure 6 is an example of a correlated subquery. The processing is such that the query matches the rows in the Lookup table for a particular Key in the Fact table. Once a match is found, we select all rows from Lookup table for that particular Key. This process repeats for each row in the Fact table.



```
SELECT Order_Id,
       Order_Customer_id
       sum(Sales_Amt)
FROM Sales X
WHERE Sales_Amt > ( SELECT AVG(Sales_Amt)
                    FROM SALES Y
                    WHERE Y.Order_Id = X.Order_ID )
```

1 Match the rows in Y for a particular Order_ID in X
2 Compute the average sales amount for a particular Order_Id
3 Repeat 1 and 2 for each row in X

Figure 6: Correlated Subquery Example

Decorrelation allows for bulk join processing, via HASH join exploitation, instead of a row at a time. There are several scenarios which are decorrelated such as:

- UPDATE statements with correlated subqueries in the SET statement
- CASE expressions containing correlated subqueries
- OR predicates containing correlated subqueries
- Subqueries containing GROUP BY

The corresponding Db2 Query Rewrite generated decorrelated query for the correlated subquery example in Figure 6 is as follows:

```
SELECT F.key
FROM fact F LEFT OUTER JOIN lookup L
  ON F.key = L.key
WHERE F.flag = 1 OR
      F.key IS NOT NULL;
```

A nested-loop join is the join method that best deals with true correlated subqueries, which can now also be chosen as of Db2 11.1 for queries where decorrelation is not possible.

Push-down of OLAP functions

SQL OLAP functions are pushed-down into the BLU engine for processing and are delivered for deeper in-database analytics with column-organized tables. These improvements include functions such as:

- RANK, ROW_NUMBER, DENSE_RANK
- FIRST_VALUE
- AVG, COUNT, COUNT_BIG
- MAX, MIN, SUM
- FIRST_VALUE, RATIO_TO_REPORT
- ROWS/RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
- ROWS BETWEEN UNBOUNDED PRECEDING AND CURENT ROW

SQL Functions Optimized for BLU

Some SQL support for BLU Columnar tables that were missing in earlier version has now been included in Db2 11.1. Which means increased performance as processing occurs natively in the BLU engine for column-organized tables. Support for column-organized tables include, wide row support, allowing you to create a table where its row length can exceed the maximum record length for the page size of the table space. Other support includes, logical character support (CODEUNITS32), European Language support (Codepage 819), IDENTITY and EXPRESSION generated columns, NOT LOGGED INITIALLY, Row and Column Access Control (RCAC) and Declared Global Temporary Table (DGTT) support, including parallel insert into not-logged DGTT from BLU source.

In addition, there are a series of String, Numeric, Date & Time functions optimized for column-organized tables as noted below:

- String Functions
  - LPAD/RPAD
  - TO_CHAR
  - INITCAP
- Numeric Functions
  - POWER, EXP, LOG10, LN
  - TO_NUMBER
  - MOD
  - SIN, COS, TAN, COT, ASIN, ACOS, ATAN
  - TRUNCATE
- Date and Time Functions
  - TO_DATE
  - MONTHNAME, DAYNAME

Automatic Dictionary Creation

Db2 11.1 improves the automatic dictionary creation of column-organized tables. The compression dictionary occurs during INSERT, INGEST, IMPORT and is now available earlier in processing. The automatic dictionary creation improvements provide better compression rates, improved query performance, increased throughput for high concurrency workloads and higher PCTENCODED (percent encoded) values in SYSCAT.COLUMNS.

3

pureScale Continuous Availability

EXTREME SCALABILITY AND AVAILABILITY

Continuous Availability

Db2 11.1 introduces some additional capabilities to Db2 pureScale that will provide simplification with installation & configuration, virtualization, workload management, manageability and additional Disaster Recovery (DR) configurations.

Db2 pureScale has been providing scalability and continuous availability through planned and unplanned events since version 9.8. Transparent scalability means that as Members are added to the Db2 pureScale Instance, applications can scale without any changes or awareness at the application level. Both scalability and availability have been designed into the architecture and possible using the highly reliable cluster caching facility (CF) - capability ported from the gold standard of availability, Db2 zOS Sysplex. Before we get into the new capabilities delivered in Db2 11.1, let's take a quick review of the pureScale architecture.

pureScale Architecture

The architecture of a Db2 pureScale cluster (also referred to as a pureScale instance), as shown in Figure 7, is an active/active data-sharing environment in which multiple nodes – called Db2 members – handle the transactional workload of the system and they also have equal and shared access to a single copy of the database on disk.

A member is a logical component and essentially a single instance of Db2 Server with its own bufferpool, memory region, log files, etc. The single copy of the database as well as the log files for each member are stored within the same shared storage using the highly scalable IBM Spectrum Scale (GPFS) filesystem.

Clients can connect into any of the members and need only know of one as there are default workload balancing capabilities that will automatically distribute the workload across all the active members. If a member fails, client connections are automatically rerouted to healthy members. A critical element of the architecture is that there is no member to member communications.
The members communicate with the cluster Caching Facility (CF) for centralized locking and cache management. For availability, there is

support for two CFs, a primary and standby. This communication takes place over a high speed, low latency interconnect via the RDMA (Remote Direct Memory Access) protocol. Alternatively, the use of TCP/IP sockets for the interconnect instead of an RDMA-based interconnect is also supported. The Db2 Cluster Services (CS) provides integrated failure detection, recovery automation and the clustered file system.

All the components are fully integrated into Db2 pureScale, in that they are all installed together as part of a single installation process, they are configured as part of installation, and they are managed and maintained together all within pureScale.

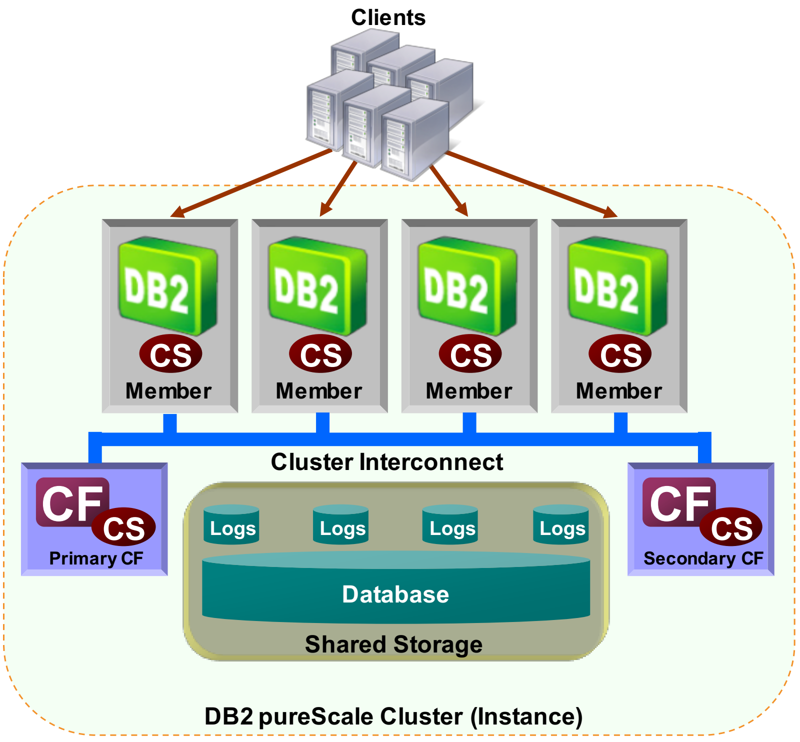


Figure 7: pureScale Architecture

Deployments

Db2 pureScale architecture allows for highly flexible deployment topologies due to the logical aspects of the members and CFs. This allows for the members and CFs to be deployed in any number of combinations, be it virtual (LPAR, VM, KVM) or physical servers. Following are some example deployment options for members and CF. For the examples, we refer to 'server' which represents either virtual or physical implementation.

The minimal requirement for pureScale is one member and one CF. However, the recommended deployment configuration, as shown in Figure 8, is two members and two CFs running across two servers. As this provides the continuously available configuration for which pureScale is architected.

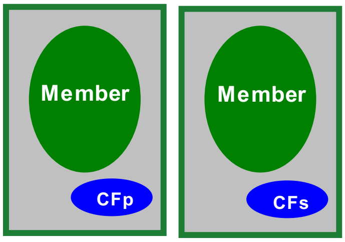


Figure 8: Two Servers - Two Members Two CFs

We can take the same two members and two CF configuration and deploy it over three servers, as in Figure 9, or four servers as in Figure 10.

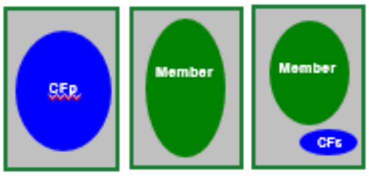


Figure 9: Three Servers - Two Members Two CFs

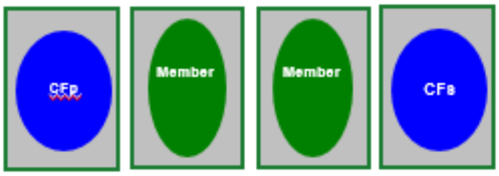


Figure 10: Four Servers - Two Members Two CFs

With three servers, you can also deploy three members and two CFs, as shown in Figure 11 and with four servers, you can have four members and two CFs (Figure 12):

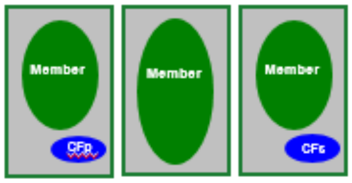


Figure 11: Three Servers - Three Members Two CFs

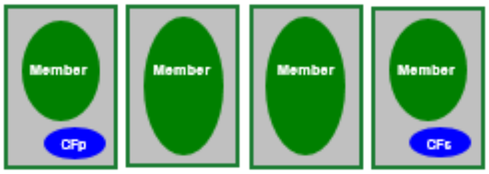


Figure 12: Four Servers - Four Members Two CFs

The decision on which deployment to utilize comes down to resources on the server and impact on server failure. Taking Figure 8 and Figure 9 as an example, a server failure in Figure 8 would result in one member and one CF going offline and the workload continues on the existing member and CF. However, in Figure 9 with three servers, a failure would result in either a Member or CF going offline or both a member and CF. If it's only the primary CF, then both member continue to process transactions with the same processing capacity. If the member or the member and CF server fail, then the workload continues on the existing member and CF the same as the Figure 8 failure example.

As you increase the number of members you increase the scalability provided by Db2 pureScale and minimize impact during a failure scenario.

In addition to the fully active/active Db2 pureScale deployments, you can also deploy pureScale using an 'active/passive licensing model', as shown earlier in Figure 3 in Chapter 1. The 'active/passive' is from the application workload perspective and not the continuously available aspect, as this remains. All application workloads are directed to the primary active member(s), sometimes referred to as the "primary" member while utilities and administrative tasks are allowed on the secondary member. This is configuration is ideal for off-loading backups and other administrative tasks from primary members as well as ensuring continuous available in the event of a member failure.

Virtualization

Db2 pureScale has supported a variety of virtualization technologies such as IBM PowerVM, PowerKVM, Red Hat KVM and VMWare ESXi. Db2 11.1 enhances the virtualization support for VMWare by now delivering RDMA over Converged Ethernet (RoCE) support in VMWare through RoCE Single-Root I/O Virtualization (SR-IOV) for RHEL 7.2. SR-IOV standard enables one PCI Express (PCIe) adapter to be presented as multiple separate logical devices (Virtual Functions) to virtual machines. This allows the virtual machines to run native RoCE and achieve near wire speed performance. This capability can be enable with Db2 pureScale when using Mellanox ConnectX-3/ConnectX-3 Pro/Connect X-3 VPI adapters for Ethernet.

Installation

The Db2 pureScale installation process in Db2 11.1 has been simplified with smarter defaults, intuitive options, parallel and quick pre-deployment validation across hosts. Allowing users to be up and running within hours of starting the installation process. There has been re-engineering efforts in install which reduces the complexity by at least 40% for sockets and takes a 30-step native GPFS setup down to a simple 4-step Db2 install process.

Db2 11.1 provides simplification in the setup and configuration of GPFS replication. GPFS replication is used for customers who want to provide protection against a complete SAN storage subsystem failure. The GPFS replication, Figure 13, is completed with a few db2cluster commands and is defined across two separate storage subsystems.

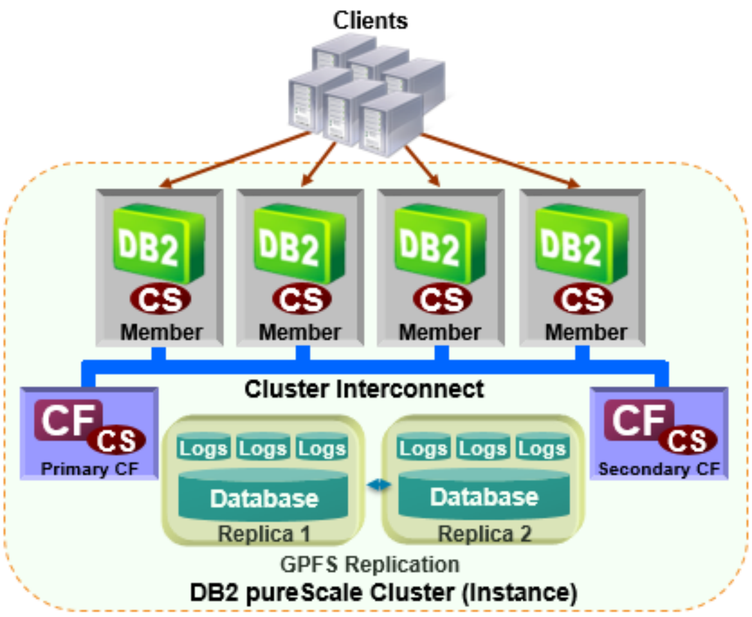


Figure 13: Four Servers - Four Members Two CFs and GPFS Replication

Workload Balancing

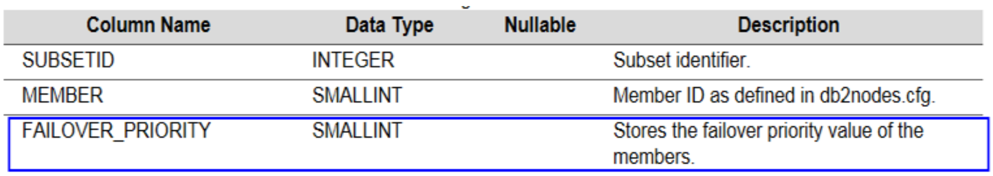
There has been additional capability included in Db2 pureScale with workload balancing when using member subsets. Member subsets allow clients to connect and be balanced across a subset of the members comprising the Db2 pureScale instance.

Db2 11.1 allows you to explicitly define the members to be used as alternates in the event of primary member failures. An alternate member can be any other member of the pureScale Instance. An alternate member of a subset behaves as any other member and processing transactions but is considered dormant for that subset until a failure occurs to a primary member of the subset. Upon a failure, the alternate member will be assigned workloads for the respective subset in addition to other workloads that has been assigned/running on that member. Db2 will attempt to maintain the same number of primary members in the respective subset, if there are enough alternates available.

Alternate members are assigned to a subset using the newly added FAILOVER_PRIORITY option of the WLM_ALTER_MEMBER_SUBSET Db2 procedure.

As part of this enhancement, a new column, FAILOVER_PRIORITY, was added to the SYSCAT.MEMBERSUBSETMEMBERS system catalog view, as seen in Table 1, to allow you to quickly see which members are primary, how many and ordering of alternate members for each subset defined.

Table 1: SYSCAT.MEMBERSUBSETMEMBERS Catalog View



Column Name	Data Type	Nullable	Description
SUBSETID	INTEGER		Subset identifier.
MEMBER	SMALLINT		Member ID as defined in db2nodes.cfg.
FAILOVER_PRIORITY	SMALLINT		Stores the failover priority value of the members.

Failover priority has values from 0-254. Members with failover priority of 0, the default if not specified, are considered primary members of the subset. Members with failover priority of 1-254 are considered alternate members of the subset. If a primary member fails, an alternate member is automatically selected to service the member subset workload. The determination of which alternate member to assign to the subset is made by choosing the alternate member with the lowest failover priority. If two alternate members have the same priority defined, then the alternate member with the lowest member number is chosen. If a primary member fails, an alternate member is automatically selected to service the member subset workload, from a lower failover priority. Creating and altering member subsets is a Db2 server side dynamically managed online operation.

Assuming we already have a member subset over members 0 and 1 called BATCH, the following example assigns member 2 and 4 as an alternate member with failover priority of 1 and 2, respectively, to the subset with name 'BATCH':

```
CALL SYSPROC.WLM_ALTER_MEMBER_SUBSET
('BATCH', NULL,
'(ADD 2 FAILOVER_PRIORITY 1,
  ADD 4 FAILOVER PRIORITY 2)');
```

The following query can be used to view the alternate member information for subsets:

```
SELECT CAST(SUBSETNAME AS CHAR(10)) AS SUBSETNAME,
       MEMBER, FAILOVER_PRIORITY
FROM SYSCAT.MEMBERSUBSETS ms,
     SYSCAT.MEMBERSUBSETMEMBERS msm
WHERE ms.SUBSETID = msm.SUBSETID;
```

Result of the above query for the BATCH example given:

```
SUBSETNAME MEMBER FAILOVER_PRIORITY
---------- ------ -----------------
BATCH           0                 0
BATCH           1                 0
BATCH           2                 1
BATCH           3                 2
```

In addition to the catalog view, the db2pd command can be used to see the alternate member information:

```
db2pd –db <dbname> –membersubsetstatus
```

To preserve the prior Db2 11.1 behavior of member subsets, during the upgrade, the value of FAILOVER_PRIORITY attribute is set to 0 for all members. Prior member subset behavior can also be maintained for newly created subsets in Db2 11.1 pureScale, by setting a FAILOVER_PRIORTY of 0 for all the members in the respective subset.

The Db2 11.1 enhancement to member subsets allows customers currently using client affinity to move their configuration to using member subsets and failover priority so that they can exploit the new benefits such as dynamic server side reconfiguration. Simplification to setting up client affinity with having control at the Server vs client – no need to db2dsdriver.cfg.

Failover behavior

During a planned or unplanned event to a member of a subset, the application will automatically be routed to the alternate member with the lowest failover priority and lowest member number, when multiple alternates are defined with the same failover priority. Should a subsequent event occur, applications will be routed to the next alternate member.

If the member subset is defined as an inclusive subset and there are not enough alternate members to failover, then the application uses the other available members in the cluster that are not part of the subset

Failback behavior

Once a primary member or a member with lower failover priority becomes available, applications from the member with the highest failover priority fail back. This means that all new applications connect to the member with lower priority that became available and existing applications connected to the alternate member, with higher failover priority, can complete processing before failing back to the lower priority member.

Manageability

Pre-Installation

As part of the improvements to installation, Db2 11.1 adds a new option, adapter list, to the db2prereqcheck command. This option is used to verify the network connectivity that all the hosts in the Db2 pureScale configuration are pingable using RDMA. The option requires an input file to be specified which contains the list of hostname, netname and adapter names for each host to be verified.

```
db2prereqcheck –adapter_list <adapter_list_filename>
```

Where <adapter_list_filename> specifies the file name that contains the list of hostname, netname, and adapter names for each of the host to be verified.

The input file must have the following format:

Table 2: Adapter list filename example

#Hostname	Netname	Interface-Adapter
hostname1	netname1	devicename-1
hostname2	netname2	devicename-2
hostname3	netname3	devicename-3

Lines that are preceded by # are considered comments and skipped.

Post-Installation

In addition to the enhancements to pre-installation above, Db2 11.1 delivers a post-installation unified health check option to the db2cluster command. The db2cluster –verify command performs a comprehensive list of checks to validate the health of the Db2 pureScale cluster. Accordingly, an alert is raised for each failed criterion and is displayed in the instance monitoring command db2instance -list. The validations performed include, but are not limited to, the following:

- Configuration settings in peer domain and GPFS cluster
- Communications between members and CFs (including RDMA)
- Replication setting for each file system
- Status of each disk in the file system

Disaster Recovery

There have been enhancements to Db2 pureScale in HADR (High Availability Disaster Recovery) and GDPC (Geographically Dispersed Db2 pureScale Cluster). Both provide zero data loss Disaster Recovery solutions integrated and delivered as part of Db2 pureScale.

HADR

Db2 11.1 enables HADR synchronous (SYNC) and near-synchronous (NEARSYNC) support to pureScale deployments. This enhancement combines the continuous availability of the Db2 pureScale feature with the robust disaster recovery capabilities of HADR providing an integrated zero data loss (i.e. RPO=0) disaster recovery solution.

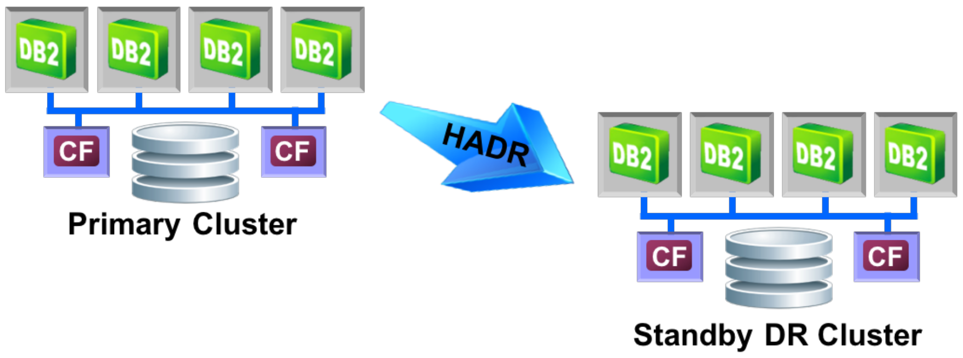


Figure 14: HADR and pureScale

The complete HADR features supported with pureScale in Db2 11.1 include:

- SYNC, NEARSYNC, ASYNC and SUPERASYNC modes
- Time delayed apply
- Log spooling
- Both non-forced (role switch) and forced (failover) takeovers

Geographically Dispersed Db2 pureScale Cluster (GDPC)

Db2 11.1 adds improved high availability for Geographically dispersed Db2 pureScale clusters (GDPC) for both RoCE & TCP/IP. pureScale allows for multiple adapter ports per member and CF to support higher bandwidth and improved redundancy at the adapter level.

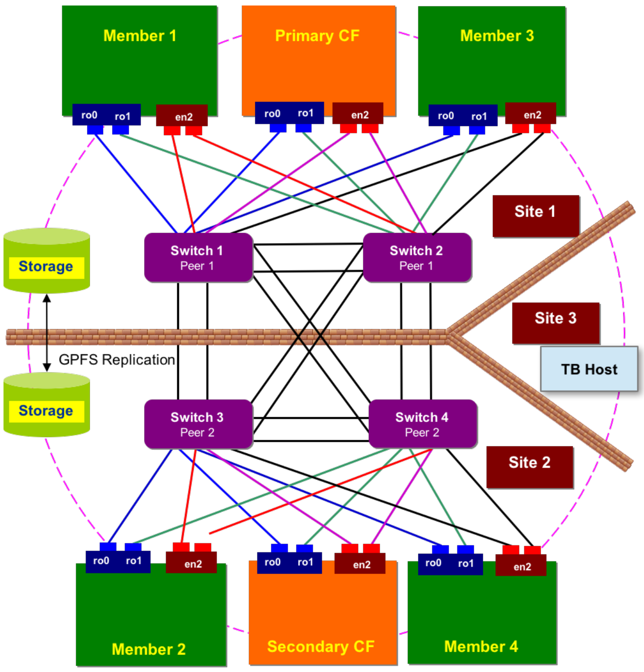


Figure 15: Geographically Dispersed Db2 pureScale Cluster

In addition, dual switches can be configured at each site in a GDPC configuration to provide greater availability in the event of a switch failure.

In Db2 11.1 a pureScale GDPC configuration exhibits the equivalent failure behavior to a non-GDPC pureScale cluster. In addition, the four-switch configuration can sustain additional concurrent failure scenarios, such as any two or 3 switch failures, without an outage.

4

Compatibility Features

Making it Easier to Move from One Database System to Another

Compatibility Features

Moving from one database vendor to another can sometimes be difficult due to syntax differences between data types, functions, and language elements. Db2 already has a high degree of compatibility with Oracle PLSQL along with some of the Oracle data types.

Db2 11.1 introduces some additional data type and function compatibility that will reduce the migration effort required when porting from other systems. There are some specific features within Db2 that are targeted at Netezza SQL and that is discussed in a separate section.

Outer Join Operator

Db2 allows the use of the Oracle outer-join operator when Oracle compatibility is turned on within a database. In Db2 11.1, the outer join operator is available by default and does not require the DBA to turn on Oracle compatibility.

Db2 supports standard join syntax for LEFT and RIGHT OUTER JOINS. However, there is proprietary syntax used by Oracle employing a keyword: "(+)" to mark the "null-producing" column reference that precedes it in an implicit join notation. The (+) appears in the WHERE clause and refers to a column of the inner table in a left outer join.

For instance:

```
SELECT * FROM T1, T2
WHERE T1.C1 = T2.C2 (+);
```

Is the same as:

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C2;
```

In the following example, we get a list of departments and their employees, as well as the names of departments who have no employees.

This example uses the standard Db2 outer join syntax.

```
SELECT DEPTNAME, LASTNAME FROM
  DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
  ON D.DEPTNO = E.WORKDEPT;
```

The results from the query are:

```
DEPTNAME             LASTNAME
-------------------- ---------------
SPIFFY COMPUTER SERV HAAS
PLANNING             THOMPSON
INFORMATION CENTER   KWAN
SUPPORT SERVICES     GEYER
MANUFACTURING SYSTEM STERN
ADMINISTRATION SYSTE PULASKI
OPERATIONS           HENDERSON
SOFTWARE SUPPORT     SPENSER
SPIFFY COMPUTER SERV LUCCHESSI
SPIFFY COMPUTER SERV O'CONNELL
```

This example works in the same manner as the previous one, but uses the "(+)" sign syntax. The format is a lot simpler to remember than OUTER JOIN syntax, but it is not part of the SQL standard.

```
SELECT DEPTNAME, LASTNAME FROM
  DEPARTMENT D, EMPLOYEE E
  WHERE D.DEPTNO = E.WORKDEPT (+);
```

CHAR datatype size increase

The CHAR datatype was limited to 254 characters in prior releases of Db2. In Db2 11.1, the limit has been increased to 255 characters to bring it in line with other SQL implementations.

```
CREATE TABLE LONGER_CHAR
  (
  NAME CHAR(255)
  );
```

Binary Data Types

Db2 11.1 introduces two new binary data types: BINARY and VARBINARY. These two data types can contain any combination of characters or binary values and are not affected by the codepage of the server that the values are stored on.

A BINARY data type is fixed and can have a maximum length of 255 bytes, while a VARBINARY column can contain up to 32,672 bytes. Each of these data types is compatible with columns created with the FOR BIT DATA keyword.

The BINARY data type will reduce the amount of conversion required from other data bases. Although binary data was supported with the FOR BIT DATA clause on a character column, it required manual DDL changes when migrating a table definition.

This example shows the creation of the three types of binary data types.

```
CREATE TABLE HEXEY
  (
  AUDIO_SHORT BINARY(255),
  AUDIO_LONG  VARBINARY(1024),
  AUDIO_CHAR  VARCHAR(255) FOR BIT DATA
  );
```

Inserting data into a binary column can be done using BINARY functions, or the use of X'xxxx' modifiers when using the VALUE clause. For fixed strings, you use the X'00' format to specify a binary value and BX'00' for variable length binary strings. For instance, the following SQL will insert data into the previous table that was created.

```
INSERT INTO HEXEY VALUES
  (BINARY('Hello there'),
   BX'2433A5D5C1',
   VARCHAR_BIT_FORMAT(HEX('Hello there')));
```

Handling binary data with a FOR BIT DATA column was sometimes tedious, so the BINARY columns will make coding a little simpler. You can compare and assign values between any of these types of columns. The next SQL statement will update the AUDIO_CHAR column with the contents of the AUDIO_SHORT column. Then the SQL will test to make sure they are the same value.

```
UPDATE HEXEY
  SET AUDIO_CHAR = AUDIO_SHORT;

SELECT COUNT(*) FROM HEXEY WHERE
  AUDIO_SHORT = AUDIO_CHAR;

1
-----------
          1
```

❶Boolean Data Types

The Boolean data type (true/false) has been available in SQLPL and PL/SQL scripts for some time. However, the Boolean data type could not be used in a table definition. Db2 11.1.1.1 now allows you to use this data type in a table definition and use TRUE/FALSE clauses to compare values. This simple table will be used to demonstrate how BOOLEAN types can be used.

```
CREATE TABLE TRUEFALSE (
    EXAMPLE INT,
    STATE   BOOLEAN
);
```

The keywords for a true value are TRUE, 'true', 't', 'yes', 'y', 'on', and '1'. For false the values are FALSE, 'false', 'f', 'no', 'n', and '0'.

```
INSERT INTO TRUEFALSE VALUES
  (1, TRUE),
  (2, FALSE),
  (3, 0),
  (4, 't'),
  (5, 'no');
```

Now we can check to see what has been inserted into the table.

```
EXAMPLE     STATE
----------- ------
          1      1
          2      0
          3      0
          4      1
          5      0
```

Retrieving the data in a SELECT statement will return an integer value for display purposes. 1 is true and 0 is false (binary 1 and 0).

Comparison operators with BOOLEAN data types will use TRUE, FALSE, 1 or 0 or any of the supported binary values. You have the choice of using the equal (=) operator or the IS or IS NOT syntax as shown in the following SQL.

```
SELECT * FROM TRUEFALSE
  WHERE STATE = TRUE OR STATE = 1 OR STATE = 'on'
        OR STATE IS TRUE;
```

Synonym Data types

Db2 has the standard data types that most developers are familiar with, like CHAR, INTEGER, and DECIMAL. There are other SQL implementations that use different names for these data types, so Db2 11.1 now allows these data types as synonyms for the base types.

These new data types are:

Table 3: Data Type Synonyms

Type	Db2 Equivalent
INT2	SMALLINT
INT4	INTEGER
INT8	BIGINT
FLOAT4	REAL
FLOAT8	FLOAT

The following SQL will create a table with these data types.

```
CREATE TABLE SYNONYM_EMPLOYEE
  (
  NAME VARCHAR(20),
  SALARY      INT4,
  BONUS       INT2,
  COMMISSION INT8,
  COMMISSION_RATE FLOAT4,
  BONUS_RATE FLOAT8
  );
```

When you create a table with these other data types, Db2 does not use these "types" in the system catalog. What Db2 will do is use the Db2 type instead of these synonym types. If you describe the contents of a table, you will see the Db2 types displayed, not these synonym types.

```
DESCRIBE TABLE SYNONYM_EMPLOYEE;

COLNAME         TYPESCHEMA TYPENAME LENGTH SCALE NULLABLE
--------------- ---------- -------- ------ ----- --------
NAME            SYSIBM     VARCHAR      20     0 Y
SALARY          SYSIBM     INTEGER       4     0 Y
BONUS           SYSIBM     SMALLINT      2     0 Y
COMMISSION      SYSIBM     BIGINT        8     0 Y
COMMISSION_RATE SYSIBM     REAL          4     0 Y
BONUS_RATE      SYSIBM     DOUBLE        8     0 Y
```

Function Name Compatibility

Db2 has a wealth of built-in functions that are equivalent to those used by other database systems, but with a different name. In Db2 11.1, these alternate function names are mapped to an equivalent Db2 function so that no re-write of the function name required.

The following table lists the function names and the equivalent Db2 function that they are mapped to.

Table 4: Function Synonyms

Function	Db2 Equivalent
COVAR_POP	COVARIANCE
STDDEV_POP	STDDEV
VAR_POP	VARIANCE
VAR_SAMP	VARIANCE_SAMP
ISNULL	IS NULL
NOTNULL	IS NOT NULL
LOG	LN
RANDOM	RAND
STRPOS	POSSTR
STRLEFT	LEFT
STRRIGHT	RIGHT
BPCHAR	VARCHAR (Casting function)
DISTRIBUTE ON	DISTRIBUTE BY

5

Netezza Compatibility

Support for Netezza SQL Syntax and Stored Procedures

Netezza Compatibility

Db2 provides features that enable applications that were written for a Netezza Performance Server (NPS) database to be used against a Db2 database.

The SQL_COMPAT global variable is used to activate the following optional NPS compatibility features:

- Double-dot notation - When operating in NPS compatibility mode, you can use double-dot notation to specify a database object.
- TRANSLATE parameter syntax - The syntax of the TRANSLATE parameter depends on whether NPS compatibility mode is being used.
- Operators - Which symbols are used to represent operators in expressions depends on whether NPS compatibility mode is being used.
- Grouping by SELECT clause columns - When operating in NPS compatibility mode, you can specify the ordinal position or exposed name of a SELECT clause column when grouping the results of a query.
- Routines written in NZPLSQL - When operating in NPS compatibility mode, the NZPLSQL language can be used in addition to the SQL PL language.

Special Operator Characters

Db2 and Netezza behave differently when using special operator characters. The table below lists the special characters and how each product interprets them.

Table 5: Db2 Special Characters

Character	Db2 Equivalent	Netezza
I/&/~	Logical OR, AND, Complement	Logical OR, AND, Complement
^	Logical XOR	Power (also **)
#	None	Logical XOR

The following SQL will display the Db2 interpretation of the special characters.

```
SET SQL_COMPAT = 'DB2';
WITH SPECIAL(OP, DESCRIPTION, EXAMPLE, RESULT) AS
  (
  VALUES
     (' ^ ','XOR        ', '2 ^ 3    ', 2 ^ 3),
     (' # ','NONE       ', '         ',0)
  )
SELECT * FROM SPECIAL;

OP   DESCRIPTION EXAMPLE   RESULT
---- ----------- --------- -----------
^    XOR         2 ^ 3               1
#    NONE                            0
```

If we turn on NPS compatibility, the ^ operator becomes a "power" operator, and the # becomes an XOR operator.

```
SET SQL_COMPAT = 'NPS';
WITH SPECIAL(OP, DESCRIPTION, EXAMPLE, RESULT) AS
  (
  VALUES
     (' ^ ','POWER      ', '2 ^ 3    ', 2 ^ 3),
     (' # ','XOR        ', '2 # 3    ', 2 # 3)
  )
SELECT * FROM SPECIAL;

OP   DESCRIPTION EXAMPLE   RESULT
---- ----------- --------- -----------
^    POWER       2 ^ 3               8
#    XOR         2 # 3               1
```

GROUP BY Ordinal Location

The GROUP BY command behavior also changes in NPS mode. The following SQL statement groups results using the default Db2 syntax:

```
SET SQL_COMPAT='DB2';

SELECT WORKDEPT,INT(AVG(SALARY))
  FROM EMPLOYEE
GROUP BY WORKDEPT;

WORKDEPT 2
-------- -----------
A00            70850
B01            94250
C01            77222
D11            58784
D21            51240
E01            80175
E11            45306
E21            47087
```

If you try using the ordinal location (similar to an ORDER BY clause), you will get an error message.

```
SELECT WORKDEPT, INT(AVG(SALARY))
  FROM EMPLOYEE
GROUP BY 1;

SQL0119N  An expression starting with "WORKDEPT" specified in a
SELECT clause, HAVING clause, or ORDER BY clause is not specified
in the GROUP BY clause or it is in a SELECT clause, HAVING clause,
or ORDER BY clause with a column function and no GROUP BY clause
is specified. SQLSTATE=42803
```

If NPS compatibility is turned on, then then you can use the GROUP BY clause with an ordinal location.

```
SET SQL_COMPAT='NPS';
SELECT WORKDEPT, INT(AVG(SALARY))
  FROM EMPLOYEE
GROUP BY 1;
```

TRANSLATE Function

The translate function syntax in Db2 is:

```
TRANSLATE(expression, to_string, from_string, padding)
```

The TRANSLATE function returns a value in which one or more characters in a string expression might have been converted to other characters. The function converts all the characters in *char-string-exp* in *from-string-exp* to the corresponding characters in *to-string-exp* or, if no corresponding characters exist, to the pad character specified by padding.

If no parameters are given to the function, the original string is converted to uppercase.

In NPS mode, the translate syntax is:

```
TRANSLATE(expression, from_string, to_string)
```

If a character is found in the *from* string, and there is no corresponding character in the *to* string, it is removed. If it was using Db2 syntax, the padding character would be used instead.

Note: If ORACLE compatibility is ON then the behavior of TRANSLATE is identical to NPS mode.

This first example will uppercase the string.

```
SET SQL_COMPAT = 'NPS';
VALUES TRANSLATE('Hello');

1
-----
HELLO
```

In this example, the letter 'o' will be replaced with an '1'.

```
VALUES TRANSLATE('Hello','o','1');

1
-----
Hell1
```

Note that you could replace more than one character by expanding both the "to" and "from" strings. This example will replace the letter "e" with a "2" as well as "o" with "1".

```
VALUES TRANSLATE('Hello','oe','12');

1
-----
H2ll1
```

Translate will also remove a character if it is not in the "to" list.

```
VALUES TRANSLATE('Hello','oel','12');

1
-----
H21
```

6

SQL Extensions

Extending SQL Syntax with More Powerful Features

SQL Extensions

Db2 has the ability to limit the amount of data retrieved on a SELECT statement through the use of the FETCH FIRST n ROWS ONLY clause. In Db2 11.1, the ability to offset the rows before fetching was added to the FETCH FIRST clause.

Simple SQL with Fetch First Clause

The FETCH first clause can be used in a variety of locations in a SELECT clause. This first example fetches only 5 rows from the EMPLOYEE table.

```
SELECT LASTNAME FROM EMPLOYEE
  FETCH FIRST 5 ROWS ONLY;

LASTNAME
---------------
HAAS
THOMPSON
KWAN
GEYER
STERN
```

You can also add ORDER BY and GROUP BY clauses in the SELECT statement. Note that Db2 still needs to process all the records and do the ORDER/GROUP BY work before limiting the answer set. So, you are not getting the first 5 rows "sorted". You are getting the entire answer set sorted before retrieving just 5 rows.

```
SELECT LASTNAME FROM EMPLOYEE
  ORDER BY LASTNAME
  FETCH FIRST 5 ROWS ONLY;

LASTNAME
---------------
ADAMSON
ALONZO
BROWN
GEYER
GOUNOT
```

Here is an example with the GROUP BY statement. This first SQL statement gives us the total answer set - the count of employees by WORKDEPT.

```
SELECT WORKDEPT, COUNT(*) FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY WORKDEPT;

WORKDEPT 2
-------- -----------
A00                5
B01                1
C01                4
D11               11
D21                7
E01                1
E11                7
E21                6
```

Adding the FETCH FIRST clause only reduces the rows returned, not the rows that are used to compute the GROUPing result.

```
SELECT WORKDEPT, COUNT(*) FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY WORKDEPT
  FETCH FIRST 5 ROWS ONLY;

WORKDEPT 2
-------- -----------
A00                5
B01                1
C01                4
D11               11
D21                7
```

OFFSET Extension

The FETCH FIRST n ROWS ONLY clause can also include an OFFSET keyword. The OFFSET keyword allows you to retrieve the answer set after skipping "n" number of rows. The syntax of the OFFSET keyword is:

```
OFFSET n ROWS FETCH FIRST x ROWS ONLY
```

The OFFSET n ROWS must precede the FETCH FIRST x ROWS ONLY clause. The OFFSET clause can be used to scroll down an answer set without having to hold a cursor. For instance, you could have the first SELECT call request 10 rows by just using the FETCH FIRST clause. After that you could request the first 10 rows be skipped before retrieving the next 10 rows.

The one thing you must be aware of is that that answer set could change between calls if you use this technique of a "moving" window. If rows are updated or added after your initial query you may get different results.

This is due to the way that Db2 adds rows to a table. If there is a DELETE and then an INSERT, the INSERTed row may end up in the empty slot. There is no guarantee of the order of retrieval. For this reason, you are better off using an ORDER BY to force the ordering although this too won't always prevent rows changing positions.

Here are the first 10 rows of the employee table (not ordered).

```
SELECT LASTNAME FROM EMPLOYEE
  FETCH FIRST 10 ROWS ONLY;

LASTNAME
---------------
HAAS
THOMPSON
KWAN
GEYER
STERN
PULASKI
HENDERSON
SPENSER
LUCCHESSI
O'CONNELL
```

You can specify a zero offset to begin from the beginning.

```
SELECT LASTNAME FROM EMPLOYEE
  OFFSET 0 ROWS
  FETCH FIRST 10 ROWS ONLY;
```

Now we can move the answer set ahead by 5 rows and get the remaining 5 rows in the answer set.

```
SELECT LASTNAME FROM EMPLOYEE
  OFFSET 5 ROWS
  FETCH FIRST 5 ROWS ONLY;

LASTNAME
---------------
PULASKI
HENDERSON
SPENSER
LUCCHESSI
O'CONNELL
```

FETCH FIRST and OFFSET in SUBSELECTs

The FETCH FIRST/OFFSET clause is not limited to regular SELECT statements. You can also limit the number of rows that are used in a subselect. In this case, you are limiting the amount of data that Db2 will scan when determining the answer set.

For instance, say you wanted to find the names of the employees who make more than the average salary of the 3rd highest paid department. (By the way, there are multiple ways to do this, but this is one approach).

The first step is to determine what the average salary is of all departments.

```
SELECT WORKDEPT, AVG(SALARY) FROM EMPLOYEE
GROUP BY WORKDEPT
ORDER BY AVG(SALARY) DESC;

WORKDEPT 2
-------- ----------------------------------------
B01                                         94250
E01                                         80175
C01                                       77222.5
A00                                         70850
D11              58783.63636363636363636363636
D21                                         51240
E21              47086.66666666666666666666667
E11              45305.71428571428571428571429
```

We only want one record from this list (the third one), so we can use the FETCH FIRST clause with an OFFSET to get the value we want (Note: we need to skip 2 rows to get to the 3rd one).

```
SELECT WORKDEPT, AVG(SALARY) FROM EMPLOYEE
GROUP BY WORKDEPT
ORDER BY AVG(SALARY) DESC
OFFSET 2 ROWS FETCH FIRST 1 ROWS ONLY;

WORKDEPT 2
-------- ----------------------------------------
C01                                       77222.5
```

And here is the list of employees that make more than the average salary of the 3rd highest department in the company.

```
SELECT LASTNAME, SALARY FROM EMPLOYEE
  WHERE
    SALARY > (
       SELECT AVG(SALARY) FROM EMPLOYEE
         GROUP BY WORKDEPT
         ORDER BY AVG(SALARY) DESC
         OFFSET 2 ROWS FETCH FIRST 1 ROW ONLY
       )
ORDER BY SALARY;
```

```
LASTNAME         SALARY
---------------- -----------
GEYER               80175.00
SPENSER             86150.00
HENDERSON           89750.00
THOMPSON            94250.00
PULASKI             96170.00
KWAN                98250.00
HAAS               152750.00
```

Alternate Syntax for FETCH FIRST

The FETCH FIRST n ROWS ONLY and OFFSET clause can also be specified using a simpler LIMIT/OFFSET syntax.

The LIMIT clause and the equivalent FETCH FIRST syntax are shown below.

Table 6: Data Type Synonyms

Syntax	Equivalent
LIMIT x	FETCH FIRST x ROWS ONLY
LIMIT x OFFSET y	OFFSET y ROWS FETCH FIRST x ROWS ONLY
LIMIT y,x	OFFSET y ROWS FETCH FIRST x ROWS ONLY

We can use the LIMIT clause with an OFFSET to get the value we want from the table.

```
SELECT WORKDEPT, AVG(SALARY) FROM EMPLOYEE
GROUP BY WORKDEPT
ORDER BY AVG(SALARY) DESC
LIMIT 1 OFFSET 2;

WORKDEPT 2
-------- ------------------------------------------
C01                                         77222.5
```

And here is the list of employees that make more than the average salary of the 3rd highest department in the company. Note that the LIMIT clause specifies only the offset (LIMIT x) or the offset and limit (LIMIT y,x) when you do not use the LIMIT keyword. One would think that LIMIT x OFFSET y would translate into LIMIT x,y but that is not the case. Don't try to figure out the SQL standards reasoning behind the syntax!

```
SELECT LASTNAME, SALARY FROM EMPLOYEE
  WHERE
    SALARY > (
       SELECT AVG(SALARY) FROM EMPLOYEE
         GROUP BY WORKDEPT
         ORDER BY AVG(SALARY) DESC
         LIMIT 2,1
       )
ORDER BY SALARY;

LASTNAME        SALARY
--------------- -----------
GEYER              80175.00
SPENSER            86150.00
HENDERSON          89750.00
THOMPSON           94250.00
PULASKI            96170.00
KWAN               98250.00
HAAS              152750.00
```

Table Creation with SELECT

The CREATE TABLE statement can now use a SELECT clause to generate the definition and LOAD the data at the same time.

The syntax of the CREATE table statement has been extended with the AS (SELECT ...) WITH DATA clause:

```
CREATE TABLE <name> AS (SELECT ...) [ WITH DATA | DEFINITION ONLY]
```

The table definition will be generated based on the SQL statement that you specify. The column names are derived from the columns that are in the SELECT list and can be changed by specifying the columns names as part of the table name: EMP(X,Y,Z,...) AS (...).

For example, the following SQL will fail because a column list was not provided:

```
CREATE TABLE AS_EMP AS (SELECT EMPNO, SALARY+BONUS FROM EMPLOYEE)
DEFINITION ONLY;

SQL0153N  The statement does not include a required column list.
SQLSTATE=42908
```

You can name a column in the SELECT list or place it in the table definition.

```
CREATE TABLE AS_EMP AS (SELECT EMPNO, SALARY+BONUS AS PAY FROM
EMPLOYEE) DEFINITION ONLY;
```

You can DESCRIBE the table to see the definition.

```
DESCRIBE TABLE AS_EMP;

COLNAME TYPESCHEMA TYPENAME  LENGTH SCALE NULLABLE
------- ---------- --------- ------ ----- --------
EMPNO   SYSIBM     CHARACTER      6     0 N
PAY     SYSIBM     DECFLOAT      16     0 Y
```

The DEFINITION ONLY clause will create the table but not load any data into it. Adding the WITH DATA clause will do an INSERT of rows into the newly created table. If you have a large amount of data to load into the table, you may be better off creating the table with DEFINITION ONLY and then using LOAD or other methods to load the data into the table.

```
CREATE TABLE AS_EMP AS
  (SELECT EMPNO, SALARY+BONUS AS PAY FROM EMPLOYEE) WITH DATA;
```

The SELECT statement in the CREATE TABLE body can be very sophisticated. It can do any type of calculation or limit the data to a subset of information.

```
CREATE TABLE AS_EMP(LAST,PAY) AS
 (
 SELECT LASTNAME, SALARY FROM EMPLOYEE
    WHERE WORKDEPT='D11'
 FETCH FIRST 3 ROWS ONLY
 ) WITH DATA;
```

You can also use the OFFSET clause as part of the FETCH FIRST ONLY to get chunks of data from the original table.

```
CREATE TABLE AS_EMP(DEPARTMENT, LASTNAME) AS
  (SELECT WORKDEPT, LASTNAME FROM EMPLOYEE
     OFFSET 5 ROWS
     FETCH FIRST 10 ROWS ONLY
  ) WITH DATA;
```

CREATE FUNCTION for Aggregate UDFs

The updated CREATE FUNCTION statement allows you to create your own aggregation functions. An aggregate function returns a single value that is the result of an evaluation of a set of like values, such as those in a column within a set of rows. Functions like SUM, AVG, and COUNT are examples of built-in column functions.

Aggregation functions can be created with a variety of programming languages and must cover four stages of the aggregation process.

- INITIATE – Called when before the SQL returns individual rows. The initialization step is used to zero out counters and other values that are used in the aggregation process.
- ACCUMULATE – As each row is retrieved, the accumulate step is called to increment or process the value in some way.
- MERGE – In a DPF (partitioned) environment, the data from each partition needs to be consolidated to come up with a final result.
- FINALIZE – The finalize step will compute the final value that the function should return to the SQL statement.

7

Date Functions

More Ways to Extract and Manipulate Date Fields

Date Functions

There are plenty of new date and time functions found in Db2 11.1. These functions allow you to extract portions from a date and format the date in a variety of different ways. While Db2 already has a number of date and time functions, these new functions allow for greater compatibility with other database implementations, making it easier to port to Db2.

Extract Function

The EXTRACT function extracts an element from a date/time value. The syntax of the EXTRACT command is:

```
EXTRACT( element FROM expression )
```

This is a slightly different format from most functions that you see in Db2. Element must be one of the following values:

Table 7: Date Elements

Element Name	Description
EPOCH	Number of seconds since 1970-01-01 00:00:00.00. The value can be positive or negative.
MILLENNIUM(S)	The millennium is to be returned.
CENTURY(CENTURIES)	The number of full 100-year periods represented by the year.
DECADE(S)	The number of full 10-year periods represented by the year.
YEAR(S)	The year portion is to be returned.
QUARTER	The quarter of the year (1 - 4) is to be returned.
MONTH	The month portion is to be returned.
WEEK	The number of the week of the year (1 - 53) that the specified day is to be returned.
DAY(S)	The day portion is to be returned.
DOW	The day of the week that is to be returned. Note that "1" represents Sunday.
DOY	The day (1 - 366) of the year that is to be returned.
HOUR(S)	The hour portion is to be returned.
MINUTE(S)	The minute portion is to be returned.
SECOND(S)	The second portion is to be returned.
MILLISECOND(S)	The second of the minute, including fractional parts to one thousandth of a second
MICROSECOND(S)	The second of the minute, including fractional parts to one millionth of a second

This SQL will return every possible extract value from the current date. The NOW keyword is a synonym for CURRENT TIMESTAMP.

```
WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('EPOCH',              EXTRACT( EPOCH          FROM NOW )),
    ('MILLENNIUM(S)',      EXTRACT( MILLENNIUM     FROM NOW )),
    ('CENTURY(CENTURIES)', EXTRACT( CENTURY        FROM NOW )),
    ('DECADE(S)',          EXTRACT( DECADE         FROM NOW )),
    ('YEAR(S)',            EXTRACT( YEAR           FROM NOW )),
    ('QUARTER',            EXTRACT( QUARTER        FROM NOW )),
    ('MONTH',              EXTRACT( MONTH          FROM NOW )),
    ('WEEK',               EXTRACT( WEEK           FROM NOW )),
    ('DAY(S)',             EXTRACT( DAY            FROM NOW )),
    ('DOW',                EXTRACT( DOW            FROM NOW )),
    ('DOY',                EXTRACT( DOY            FROM NOW )),
    ('HOUR(S)',            EXTRACT( HOURS          FROM NOW )),
    ('MINUTE(S)',          EXTRACT( MINUTES        FROM NOW )),
    ('SECOND(S)',          EXTRACT( SECONDS        FROM NOW )),
    ('MILLISECOND(S)',     EXTRACT( MILLISECONDS   FROM NOW )),
    ('MICROSECOND(S)',     EXTRACT( MICROSECONDS   FROM NOW ))
  )
SELECT * FROM DATES;
```

```
FUNCTION           RESULT
------------------ ---------------------------
EPOCH                        1474090894.000000
MILLENNIUM(S)                         2.000000
CENTURY(CENTURIES)                   20.000000
DECADE(S)                           201.000000
YEAR(S)                            2016.000000
QUARTER                               3.000000
MONTH                                 9.000000
WEEK                                 38.000000
DAY(S)                               17.000000
DOW                                   7.000000
DOY                                 261.000000
HOUR(S)                               5.000000
MINUTE(S)                            41.000000
SECOND(S)                            34.578000
MILLISECOND(S)                    34578.000000
MICROSECOND(S)                 34578000.000000
```

DATE_PART Function

DATE_PART is like the EXTRACT function but it uses the more familiar function syntax:

```
  DATE_PART(element, expression)
```

In the case of the DATE_PART function, the element must be placed in quotes, rather than as a keyword in the EXTRACT function. in addition, the DATE_PART always returns a BIGINT, while the EXTRACT function will return a different data type depending on the element being returned. For instance, the SECONDs option for EXTRACT returns a DECIMAL result while the DATE_PART returns a truncated BIGINT.

```
 WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('EPOCH',                DATE_PART('EPOCH'           ,NOW )),
    ('MILLENNIUM(S)',        DATE_PART('MILLENNIUM'      ,NOW )),
    ('CENTURY(CENTURIES)', DATE_PART('CENTURY'         ,NOW )),
    ('DECADE(S)',            DATE_PART('DECADE'          ,NOW )),
    ('YEAR(S)',              DATE_PART('YEAR'            ,NOW )),
    ('QUARTER',              DATE_PART('QUARTER'         ,NOW )),
    ('MONTH',                DATE_PART('MONTH'           ,NOW )),
    ('WEEK',                 DATE_PART('WEEK'            ,NOW )),
    ('DAY(S)',               DATE_PART('DAY'             ,NOW )),
    ('DOW',                  DATE_PART('DOW'             ,NOW )),
    ('DOY',                  DATE_PART('DOY'             ,NOW )),
    ('HOUR(S)',              DATE_PART('HOURS'           ,NOW )),
    ('MINUTE(S)',            DATE_PART('MINUTES'         ,NOW )),
    ('SECOND(S)',            DATE_PART('SECONDS'         ,NOW )),
    ('MILLISECOND(S)',       DATE_PART('MILLISECONDS'    ,NOW )),
    ('MICROSECOND(S)',       DATE_PART('MICROSECONDS'    ,NOW ))
  )
SELECT * FROM DATES;

FUNCTION           RESULT
------------------ --------------------
EPOCH                        1474090894
MILLENNIUM(S)                         2
CENTURY(CENTURIES)                   20
DECADE(S)                           201
YEAR(S)                            2016
QUARTER                               3
MONTH                                 9
WEEK                                 38
DAY(S)                               17
DOW                                   7
DOY                                 261
HOUR(S)                               5
MINUTE(S)                            41
SECOND(S)                            34
MILLISECOND(S)                    34809
MICROSECOND(S)                 34809000
```

DATE_TRUNC Function

DATE_TRUNC computes the same results as the DATE_PART function but then truncates the value down. Note that not all values can be truncated. The function syntax is:

```
  DATE_TRUNC(element, expression)
```

The element must be placed in quotes, rather than as a keyword in the EXTRACT function. Note that DATE_TRUNC always returns a BIGINT.

The elements that can be truncated are:

Table 8: DATE_TRUNC Elements

Element Name	Description
MILLENNIUM(S)	The millennium is to be returned.
CENTURY(CENTURIES)	The number of full 100-year periods represented by the year.
DECADE(S)	The number of full 10-year periods represented by the year.
YEAR(S)	The year portion is to be returned.
QUARTER	The quarter of the year (1 - 4) is to be returned.
MONTH	The month portion is to be returned.
WEEK	The number of the week of the year (1 - 53) that the specified day is to be returned.
DAY(S)	The day portion is to be returned.
HOUR(S)	The hour portion is to be returned.
MINUTE(S)	The minute portion is to be returned.
SECOND(S)	The second portion is to be returned.
MILLISECOND(S)	The second of the minute, including fractional parts to one thousandth of a second
MICROSECOND(S)	The second of the minute, including fractional parts to one millionth of a second

```
WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('MILLENNIUM(S)',      DATE_TRUNC('MILLENNIUM',NOW )),
    ('CENTURY(CENTURIES)', DATE_TRUNC('CENTURY'   ,NOW )),
    ('DECADE(S)',          DATE_TRUNC('DECADE'    ,NOW )),
    ('YEAR(S)',            DATE_TRUNC('YEAR'      ,NOW )),
    ('QUARTER',            DATE_TRUNC('QUARTER'   ,NOW )),
    ('MONTH',              DATE_TRUNC('MONTH'     ,NOW )),
    ('WEEK',               DATE_TRUNC('WEEK'      ,NOW )),
    ('DAY(S)',             DATE_TRUNC('DAY'       ,NOW )),
    ('HOUR(S)',            DATE_TRUNC('HOURS'     ,NOW )),
    ('MINUTE(S)',          DATE_TRUNC('MINUTES'   ,NOW )),
    ('SECOND(S)',          DATE_TRUNC('SECONDS'   ,NOW )),
    ('MILLISEC(S)',        DATE_TRUNC('MILLISECONDS ',NOW )),
    ('MICROSEC(S)',        DATE_TRUNC('MICROSECONDS ',NOW ))
  )
SELECT * FROM DATES;

FUNCTION           RESULT
------------------ --------------------------
MILLENNIUM(S)      2000-01-01 00:00:00.000000
CENTURY(CENTURIES) 2000-01-01 00:00:00.000000
DECADE(S)          2010-01-01 00:00:00.000000
YEAR(S)            2016-01-01 00:00:00.000000
QUARTER            2016-07-01 00:00:00.000000
MONTH              2016-09-01 00:00:00.000000
WEEK               2016-09-12 00:00:00.000000
DAY(S)             2016-09-17 00:00:00.000000
HOUR(S)            2016-09-17 05:00:00.000000
MINUTE(S)          2016-09-17 05:41:00.000000
SECOND(S)          2016-09-17 05:41:35.000000
MILLISEC(S)        2016-09-17 05:41:35.049000
MICROSEC(S)        2016-09-17 05:41:35.049000
```

Extracting Specific Days from a Month

There are three functions that retrieve day information from a date.

- DAYOFMONTH – returns an integer between 1 and 31 that represents the day of the argument
- FIRST_DAY – returns a date or timestamp that represents the first day of the month of the argument
- DAYS_TO_END_OF_MONTH – returns the number of days to the end of the month

The following examples assume that the current date is September 17th, 2016.

```
VALUES NOW;

1
--------------------------
2016-09-17 05:41:35.229000
```

DAYOFMONTH returns the day of the month.

```
VALUES DAYOFMONTH(NOW);

1
-----------
         17
```

FIRST_DAY will return the first day of the month. You could probably compute this with standard SQL date functions, but it is a lot easier just to use this built-in function.

```
VALUES FIRST_DAY(NOW);

1
--------------------------
2016-09-01 05:41:35.399000
```

Finally, DAYS_TO_END_OF_MOTNH will return the number of days to the end of the month. A Zero would be returned if you are on the last day of the month.

```
VALUES DAYS_TO_END_OF_MONTH(NOW);

1
-----------
         13
```

Date Addition Functions

The date addition functions will add or subtract days from a current timestamp. The functions that are available are:

- ADD_YEARS – Add years to a date
- ADD_MONTHS – Add months to a date
- ADD_DAYS – Add days to a date
- ADD_HOURS – Add hours to a date
- ADD_MINUTES – Add minutes to a date
- ADD_SECONDS – Add seconds to a date

The format of the function is:

```
ADD_DAYS ( expression, numeric expression )
```

The following SQL will add one "unit" to the current date.

```
WITH DATES(FUNCTION, RESULT) AS
  (
  VALUES
    ('CURRENT DATE    ',NOW),
    ('ADD_YEARS       ',ADD_YEARS(NOW,1)),
    ('ADD_MONTHS      ',ADD_MONTHS(NOW,1)),
    ('ADD_DAYS        ',ADD_DAYS(NOW,1)),
    ('ADD_HOURS       ',ADD_HOURS(NOW,1)),
    ('ADD_MINUTES     ',ADD_MINUTES(NOW,1)),
    ('ADD_SECONDS     ',ADD_SECONDS(NOW,1))
  )
SELECT * FROM DATES;

FUNCTION         RESULT
---------------- --------------------------
CURRENT DATE     2016-09-17 05:41:35.669000
ADD_YEARS        2017-09-17 05:41:35.669000
ADD_MONTHS       2016-10-17 05:41:35.669000
ADD_DAYS         2016-09-18 05:41:35.669000
ADD_HOURS        2016-09-17 06:41:35.669000
ADD_MINUTES      2016-09-17 05:42:35.669000
ADD_SECONDS      2016-09-17 05:41:36.669000
```

A negative number can be used to subtract values from the current date.

```
WITH DATES(FUNCTION, RESULT) AS
  (
  VALUES
    ('CURRENT DATE    ',NOW),
    ('ADD_YEARS       ',ADD_YEARS(NOW,-1)),
    ('ADD_MONTHS      ',ADD_MONTHS(NOW,-1)),
    ('ADD_DAYS        ',ADD_DAYS(NOW,-1)),
    ('ADD_HOURS       ',ADD_HOURS(NOW,-1)),
    ('ADD_MINUTES     ',ADD_MINUTES(NOW,-1)),
    ('ADD_SECONDS     ',ADD_SECONDS(NOW,-1))
  )
SELECT * FROM DATES;

FUNCTION         RESULT
---------------- --------------------------
CURRENT DATE     2016-09-17 05:41:35.749000
ADD_YEARS        2015-09-17 05:41:35.749000
ADD_MONTHS       2016-08-17 05:41:35.749000
ADD_DAYS         2016-09-16 05:41:35.749000
ADD_HOURS        2016-09-17 04:41:35.749000
ADD_MINUTES      2016-09-17 05:40:35.749000
ADD_SECONDS      2016-09-17 05:41:34.749000
```

Extracting Weeks, Months, Quarters, and Years from a Date

There are four functions that extract different values from a date. These functions include:

- THIS_QUARTER – returns the first day of the quarter
- THIS_WEEK – returns the first day of the week (Sunday is considered the first day of that week)
- THIS_MONTH – returns the first day of the month
- THIS_YEAR – returns the first day of the year

The following SQL will return all the current (THIS) values of the current date.

```
WITH DATES(FUNCTION, RESULT) AS
  (
  VALUES
    ('CURRENT DATE    ',NOW),
    ('THIS_WEEK       ',THIS_WEEK(NOW)),
    ('THIS_MONTH      ',THIS_MONTH(NOW)),
    ('THIS_QUARTER    ',THIS_QUARTER(NOW)),
    ('THIS_YEAR       ',THIS_YEAR(NOW))
  )
SELECT * FROM DATES;
```

```
FUNCTION         RESULT
---------------- --------------------------
CURRENT DATE     2016-09-17 05:41:35.879000
THIS_WEEK        2016-09-11 00:00:00.000000
THIS_MONTH       2016-09-01 00:00:00.000000
THIS_QUARTER     2016-07-01 00:00:00.000000
THIS_YEAR        2016-01-01 00:00:00.000000
```

There is also a NEXT function for each of these. The NEXT function will return the next week, month, quarter, or year given a current date.

```
WITH DATES(FUNCTION, RESULT) AS
  (
  VALUES
    ('CURRENT DATE    ',NOW),
    ('NEXT_WEEK       ',NEXT_WEEK(NOW)),
    ('NEXT_MONTH      ',NEXT_MONTH(NOW)),
    ('NEXT_QUARTER    ',NEXT_QUARTER(NOW)),
    ('NEXT_YEAR       ',NEXT_YEAR(NOW))
  )
SELECT * FROM DATES;

FUNCTION         RESULT
---------------- --------------------------
CURRENT DATE     2016-09-17 05:41:35.979000
NEXT_WEEK        2016-09-18 00:00:00.000000
NEXT_MONTH       2016-10-01 00:00:00.000000
NEXT_QUARTER     2016-10-01 00:00:00.000000
NEXT_YEAR        2017-01-01 00:00:00.000000
```

Next Day Function

The previous set of functions returned a date value for the current week, month, quarter, or year (or the next one if you used the NEXT function). The NEXT_DAY function returns the next day (after the date you supply) based on the string representation of the day. The date string will be dependent on the codepage that you are using for the database.

The date (from an English perspective) can be:

Table 9: Days and Short Form

Day	Short form
Monday	MON
Tuesday	TUE
Wednesday	WED
Thursday	THU
Friday	FRI
Saturday	SAT
Sunday	SUN

The following SQL will show you the "day" after the current date that is Monday through Sunday.

```
WITH DATES(FUNCTION, RESULT) AS
  (
  VALUES
    ('CURRENT DATE    ',NOW),
    ('Monday          ',NEXT_DAY(NOW,'Monday')),
    ('Tuesday         ',NEXT_DAY(NOW,'TUE')),
    ('Wednesday       ',NEXT_DAY(NOW,'Wednesday')),
    ('Thursday        ',NEXT_DAY(NOW,'Thursday')),
    ('Friday          ',NEXT_DAY(NOW,'FRI')),
    ('Saturday        ',NEXT_DAY(NOW,'Saturday')),
    ('Sunday          ',NEXT_DAY(NOW,'Sunday'))
  )
SELECT * FROM DATES;

FUNCTION          RESULT
---------------   --------------------------
CURRENT DATE      2016-09-17 05:41:36.139000
Monday            2016-09-19 05:41:36.139000
Tuesday           2016-09-20 05:41:36.139000
Wednesday         2016-09-21 05:41:36.139000
Thursday          2016-09-22 05:41:36.139000
Friday            2016-09-23 05:41:36.139000
Saturday          2016-09-24 05:41:36.139000
Sunday            2016-09-18 05:41:36.139000
```

Between Date/Time Functions

These date functions compute the number of full seconds, minutes, hours, days, weeks, and years between two dates. If there isn't a full value between the two objects (like less than a day), a zero will be returned. These new functions are:

- HOURS_BETWEEN – returns the number of full hours between two arguments
- MINUTES_BETWEEN – returns the number of full minutes between two arguments
- SECONDS_BETWEEN – returns the number of full seconds between two arguments
- DAYS_BETWEEN – returns the number of full days between two arguments
- WEEKS_BETWEEN – returns the number of full weeks between two arguments
- YEARS_BETWEEN – returns the number of full years between two arguments

The format of all of these functions is:

```
DAYS_BETWEEN( expression1, expression2 )
```

The following SQL will use a date that is in the future with exactly one extra second, minute, hour, day, week and year added to it.

```
CREATE VARIABLE FUTURE_DATE TIMESTAMP DEFAULT(NOW + 1 SECOND + 1
MINUTE + 1 HOUR + 8 DAYS + 1 YEAR);

WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('SECONDS_BETWEEN',SECONDS_BETWEEN(FUTURE_DATE,NOW)),
    ('MINUTES_BETWEEN',MINUTES_BETWEEN(FUTURE_DATE,NOW)),
    ('HOURS_BETWEEN  ',HOURS_BETWEEN(FUTURE_DATE,NOW)),
    ('DAYS BETWEEN   ',DAYS_BETWEEN(FUTURE_DATE,NOW)),
    ('WEEKS_BETWEEN  ',WEEKS_BETWEEN(FUTURE_DATE,NOW)),
    ('YEARS_BETWEEN  ',YEARS_BETWEEN(FUTURE_DATE,NOW))
  )
SELECT * FROM DATES;

FUNCTION        RESULT
--------------- --------------------
SECONDS_BETWEEN             32230861
MINUTES_BETWEEN               537181
HOURS_BETWEEN                   8953
DAYS BETWEEN                     373
WEEKS_BETWEEN                     53
YEARS_BETWEEN                      1
```

MONTHS_BETWEEN Function

You may have noticed that the MONTHS_BETWEEN function was not in the previous list of functions. The reason for this is that the value returned for MONTHS_BETWEEN is different from the other functions.

The MONTHS_BETWEEN function returns a DECIMAL value rather than an integer value. The reason for this is that the duration of a month is not as precise as a day, week or year. The following example will show how the duration is a decimal value rather than an integer. You could always truncate the value if you want an integer.

```
WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('0 MONTH          ',MONTHS_BETWEEN(NOW, NOW)),
    ('1 MONTH          ',MONTHS_BETWEEN(NOW + 1 MONTH, NOW)),
    ('1 MONTH + 1 DAY',MONTHS_BETWEEN(NOW + 1 MONTH + 1 DAY, NOW)),
    ('LEAP YEAR        ',MONTHS_BETWEEN('2016-02-01','2016-03-01')),
    ('NON-LEAP YEAR   ',MONTHS_BETWEEN('2015-02-01','2015-03-01'))
  )
SELECT * FROM DATES;
```

```
FUNCTION         RESULT
---------------- ----------------------------------
0 MONTH                           0.000000000000000
1 MONTH                           1.000000000000000
1 MONTH + 1 DAY                   1.032258064516129
LEAP YEAR                        -1.000000000000000
NON-LEAP YEAR                    -1.000000000000000
```

Date Duration Functions

An alternate way of representing date durations is through the use of an integer with the format YYYYMMDD where the YYYY represents the year, MM for the month and DD for the day. Date durations are easier to manipulate than timestamp values and take up substantially less storage.

There are two new functions.

- YMD_BETWEEN returns a numeric value that specifies the number of full years, full months, and full days between two datetime values
- AGE returns a numeric value that represents the number of full years, full months, and full days between the current timestamp and the argument

This SQL statement will return various AGE calculations based on the current timestamp.

```
WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('AGE+ 1 DAY                      ',AGE(NOW - 1 DAY)),
    ('AGE+ 1 MONTH                    ',AGE(NOW - 1 MONTH)),
    ('AGE+ 1 YEAR                     ',AGE(NOW - 1 YEAR)),
    ('AGE+ 1 DAY + 1 MONTH            ',AGE(NOW - 1 DAY - 1 MONTH)),
    ('AGE+ 1 DAY + 1 YEAR             ',AGE(NOW - 1 DAY - 1 YEAR)),
    ('AGE+ 1 DAY + 1 MONTH + 1 YEAR',AGE(NOW - 1 DAY - 1 MONTH - 1 YEAR))
  )
SELECT * FROM DATES;
```

```
FUNCTION                       RESULT
------------------------------ -----------
AGE + 1 DAY                              1
AGE + 1 MONTH                          100
AGE + 1 YEAR                         10000
AGE + 1 DAY + 1 MONTH                  101
AGE + 1 DAY + 1 YEAR                 10001
AGE + 1 DAY + 1 MONTH + 1 YEAR       10101
```

The YMD_BETWEEN function is like the AGE function except that it takes two date arguments. We can simulate the AGE function by supplying the NOW function to the YMD_BETWEEN function.

```
WITH DATES(FUNCTION, RESULT) AS (
  VALUES
    ('1 DAY          ',YMD_BETWEEN(NOW,NOW - 1 DAY)),
    ('1 MON          ',YMD_BETWEEN(NOW,NOW - 1 MONTH)),
    ('1 YR           ',YMD_BETWEEN(NOW,NOW - 1 YEAR)),
    ('1 DAY + 1 MON  ',YMD_BETWEEN(NOW,NOW - 1 DAY - 1 MONTH)),
    ('1 DAY +1 YR    ',YMD_BETWEEN(NOW,NOW - 1 DAY - 1 YEAR)),
    ('1 DAY+1 MON +1 YR',YMD_BETWEEN(NOW,NOW - 1 DAY - 1 MONTH - 1 YEAR))
  )
SELECT * FROM DATES;

FUNCTION                       RESULT
------------------------------ -----------
1 DAY                                    1
1 MONTH                                100
1 YEAR                               10000
1 DAY + 1 MONTH                        101
1 DAY + 1 YEAR                       10001
1 DAY + 1 MONTH + 1 YEAR             10101
```

OVERLAPS Predicate

The OVERLAPS predicate is used to determine whether two chronological periods overlap. This is not a function within Db2, but rather a special SQL syntax extension.

A chronological period is specified by a pair of date-time expressions. The first expression specifies the start of a period; the second specifies its end.

```
(start1, end1) OVERLAPS (start2, end2)
```

The beginning and end values are not included in the periods. The following summarizes the overlap logic. For example, the periods 2016-10-19 to 2016-10-20 and 2016-10-20 to 2016-10-21 do not overlap.

The following interval does not overlap.

```
VALUES
  CASE
    WHEN (NOW, NOW + 1 DAY) OVERLAPS (NOW + 1 DAY, NOW + 2 DAYS)
      THEN 'Overlaps'
    ELSE 'No Overlap'
  END;

1
----------
No Overlap
```

If the first date range is extended by one day, then the range will overlap.

```
VALUES
  CASE
    WHEN (NOW, NOW + 2 DAYS) OVERLAPS (NOW + 1 DAY, NOW + 2 DAYS)
      THEN 'Overlaps'
    ELSE 'No Overlap'
  END;

1
----------
Overlaps
```

Identical date ranges will overlap.

```
VALUES
  CASE
    WHEN (NOW, NOW + 1 DAY) OVERLAPS (NOW, NOW + 1 DAY)
      THEN 'Overlaps'
    ELSE 'No Overlap'
  END;

1
----------
Overlaps
```

UTC Time Conversions

Db2 has two functions that allow you to translate timestamps to and from UTC (Coordinated Universal Time).

- The FROM_UTC_TIMESTAMP scalar function returns a TIMESTAMP that is converted from Coordinated Universal Time to the time zone specified by the time zone string.
- The TO_UTC_TIMESTAMP scalar function returns a TIMESTAMP that is converted to Coordinated Universal Time from the timezone that is specified by the timezone string.

The format of the two functions is:

```
FROM_UTC_TIMESTAMP( expression, timezone)

TO_UTC_TIMESTAMP( expression, timezone)
```

The return value from each of these functions is a timestamp. The "expression" is a timestamp that you want to convert to the local timezone (or convert to UTC). The timezone is an expression that specifies the time zone that the expression is to be adjusted to. The value of the timezone-expression must be a time zone name from the Internet Assigned Numbers Authority (IANA) time zone database. The standard format for a time zone name in the IANA database is Area/Location, where:

- Area is the English name of a continent, ocean, or the special area 'Etc'
- Location is the English name of a location within the area; usually a city, or small island

Examples:

- "America/Toronto"
- "Asia/Sakhalin"
- "Etc/UTC" (which represents Coordinated Universal Time)

For complete details on the valid set of time zone names and the rules that are associated with those time zones, refer to the IANA time zone database. Db2 11.1 uses version 2010c of the IANA time zone database.

The result is a timestamp, adjusted from/to the Coordinated Universal Time zone to the time zone specified by the timezone-expression. If the timezone-expression returns a value that is not a time zone in the IANA time zone database, then the value of expression is returned without being adjusted.

Using UTC Functions

One of the applications for using the UTC is to take a transaction timestamp and normalize it across all systems that access the data. You can convert the timestamp to UTC on insert and then when it is retrieved, it can be converted back to the local timezone.

This example will use several techniques to hide the complexity of changing timestamps to local time zones. The following SQL will create our base transaction table (TXS_BASE) that will be used throughout the example.

```
CREATE TABLE TXS_BASE
  (
  ID INTEGER NOT NULL,
  CUSTID INTEGER NOT NULL,
  TXTIME_UTC TIMESTAMP NOT NULL
  );
```

The UTC functions will be written to take advantage of a local timezone variable called TIME_ZONE. This variable will contain the timezone of the server (or user) that is running the transaction. In this case we are using the timezone in Toronto, Canada.

```
CREATE OR REPLACE VARIABLE TIME_ZONE VARCHAR(255)
  DEFAULT('America/Toronto');
```

The SET Command can be used to update the TIME_ZONE to the current location we are in.

```
SET TIME_ZONE = 'America/Toronto';
```

To retrieve the value of the current timezone, we take advantage of a simple user-defined function called GET_TIMEZONE. It just retrieves the contents of the current TIME_ZONE variable that we set up.

```
CREATE OR REPLACE FUNCTION GET_TIMEZONE()
  RETURNS VARCHAR(255)
LANGUAGE SQL CONTAINS SQL
  RETURN (TIME_ZONE);
```

The TXS view is used by all SQL statements rather than the TXS_BASE table. The reason for this is to take advantage of INSTEAD OF triggers that can manipulate the UTC without modifying the original SQL.

Note that when the data is returned from the view that the TXTIME field is converted from UTC to the current TIMEZONE that we are in.

```
CREATE OR REPLACE VIEW TXS AS
  (
  SELECT
     ID,
     CUSTID,
     FROM_UTC_TIMESTAMP(TXTIME_UTC,GET_TIMEZONE()) AS TXTIME
  FROM
     TXS_BASE
  );
```

An INSTEAD OF trigger (INSERT, UPDATE, and DELETE) is created against the TXS view so that any insert or update on a TXTIME column will be converted back to the UTC value. From an application perspective, we are using the local time, not the UTC time.

```
CREATE OR REPLACE TRIGGER I_TXS
  INSTEAD OF INSERT ON TXS
  REFERENCING NEW AS NEW_TXS
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  INSERT INTO TXS_BASE VALUES (
     NEW_TXS.ID,
     NEW_TXS.CUSTID,
     TO_UTC_TIMESTAMP(NEW_TXS.TXTIME,GET_TIMEZONE())
     );
END
/

CREATE OR REPLACE TRIGGER U_TXS
  INSTEAD OF UPDATE ON TXS
  REFERENCING NEW AS NEW_TXS OLD AS OLD_TXS
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE TXS_BASE
     SET (ID, CUSTID, TXTIME_UTC) =
         (NEW_TXS.ID,
          NEW_TXS.CUSTID,
          TO_UTC_TIMESTAMP(NEW_TXS.TXTIME,TIME_ZONE)
          )
     WHERE
       TXS_BASE.ID = OLD_TXS.ID
     ;
END
/

CREATE OR REPLACE TRIGGER D_TXS
  INSTEAD OF DELETE ON TXS
  REFERENCING OLD AS OLD_TXS
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  DELETE FROM TXS_BASE
     WHERE
       TXS_BASE.ID = OLD_TXS.ID
     ;
END
/
```

At this point in time(!) we can start inserting records into our table. We have already set the timezone to be Toronto, so the next insert statement will take the current time (NOW) and insert it into the table.

For reference, here is the current time when the example was run.

```
VALUES NOW;

1
--------------------------
2016-09-17 05:53:18.956000
```

We will insert one record into the table and immediately retrieve the result.

```
INSERT INTO TXS VALUES(1,1,NOW);

SELECT * FROM TXS;

ID          CUSTID      TXTIME
----------- ----------- --------------------------
          1           1 2016-09-17 05:53:19.056000
```

Note that the timestamp appears to be the same as what we insert (plus or minus a few seconds). What actually sits in the base table is the UTC time.

```
SELECT * FROM TXS_BASE;

ID          CUSTID      TXTIME_UTC
----------- ----------- --------------------------
          1           1 2016-09-17 09:53:19.056000
```

We can modify the time that is returned to us by changing our local timezone. The statement will make the system think we are in Vancouver.

```
SET TIME_ZONE = 'America/Vancouver';
```

Retrieving the results will show that the timestamp has shifted by 3 hours (Vancouver is 3 hours behind Toronto).

```
SELECT * FROM TXS;

ID          CUSTID      TXTIME
----------- ----------- --------------------------
          1           1 2016-09-17 02:53:19.056000
```

So, what happens if we insert a record into the table now that we are in Vancouver?

```
INSERT INTO TXS VALUES(2,2,NOW);

SELECT * FROM TXS;

ID          CUSTID      TXTIME
----------- ----------- --------------------------
          1           1 2016-09-17 02:53:19.056000
          2           2 2016-09-17 05:53:19.436000
```

The data retrieved reflects the fact that we are now in Vancouver from an application perspective. Looking at the base table and you will see that everything has been converted to UTC time.

```
SELECT * FROM TXS_BASE;

ID          CUSTID      TXTIME_UTC
----------- ----------- --------------------------
          1           1 2016-09-17 09:53:19.056000
          2           2 2016-09-17 12:53:19.436000
```

Finally, we can switch back to Toronto time and see when the transactions were done. You will see that from a Toronto perspective that the transactions were done three hours later because of the timezone differences.

```
SET TIME_ZONE = 'America/Toronto';

SELECT * FROM TXS;

ID          CUSTID      TXTIME
----------- ----------- --------------------------
          1           1 2016-09-17 05:53:19.056000
          2           2 2016-09-17 08:53:19.436000
```

8

Binary Manipulation

New Binary and Hexadecimal Operators

Hex Functions

Several new HEX manipulation functions have been added to Db2 11.1. There are a class of functions that manipulate different size integers (SMALL, INTEGER, BIGINT) using NOT, OR, AND, and XOR. In addition to these functions, there are several functions that display and convert values into hexadecimal values.

INTN Functions

The INTN functions are bitwise functions that operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value. The function names all include the size of the integers that are being manipulated:

Table 10: Argument Sizes

Identifier (N)	Integer Size
2	SMALLINT – 2 bytes
4	INTEGER – 4 bytes
8	BIGINT – 8 bytes

There are four functions:

- INTNAND - Performs a bitwise AND operation
- INTNOR - Performs a bitwise OR operation
- INTNXOR Performs a bitwise exclusive OR operation
- INTNNOT - Performs a bitwise NOT operation

This example will show the four functions used against INT2 data types.

```
WITH LOGIC(EXAMPLE, X, Y, RESULT) AS
  (
  VALUES
     ('INT2AND(X,Y)',1,3,INT2AND(1,3)),
     ('INT2OR(X,Y) ',1,3,INT2OR(1,3)),
     ('INT2XOR(X,Y)',1,3,INT2XOR(1,3)),
     ('INT2NOT(X)  ',1,3,INT2NOT(1))   )
SELECT * FROM LOGIC;
```

```
EXAMPLE      X      Y      RESULT
------------ ------ ------ ------
INT2AND(X,Y)      1      3      1
INT2OR(X,Y)       1      3      3
INT2XOR(X,Y)      1      3      2
INT2NOT(X)        1      3     -2
```

You can mix and match the INT2, INT4, and INT8 values in these functions but you may get truncation if the value is too big.

TO_HEX Function

The TO_HEX function converts a numeric expression into a character hexadecimal representation. For example, the numeric value 255 represents x'FF'. The value returned from this function is a VARCHAR value and its length depends on the size of the number you supply.

```
VALUES
  TO_HEX(255);

1
--------
ff
```

RAWTOHEX Function

The RAWTOHEX function returns a hexadecimal representation of a value as a character string. The result is a character string itself.

```
VALUES RAWTOHEX('00');

1
----
3030
```

The string "00" converts to a hex representation of x'3030' which is 12336 in Decimal. So, the TO_HEX function would convert this back to the HEX representation.

```
VALUES
  TO_HEX(12336);

1
--------
3030
```

9

Regular Expressions

Extensive Regular Expression Support for Searching and Data Validation

Regular Expressions

Db2 11.1 introduced support for regular expressions. Regular expressions allow you to do very complex pattern matching in character strings. Normal SQL LIKE searches are limited to very specific patterns, but regular expressions have a rich syntax that gives you much more flexibility in searching.

Regular Expression Examples

The examples in this section are using a list of some of the underground stations of London Underground Central line (existing stations only, not historical ones!). This table will be used for all of the examples within this section.

```
CREATE TABLE CENTRAL_LINE
  (
  STATION_NO INTEGER GENERATED ALWAYS AS IDENTITY,
  STATION VARCHAR(31),
  UPPER_STATION VARCHAR(31) GENERATED ALWAYS AS (UCASE(STATION))
  );
```

The list of stations is:

Table 1: London Central Line (Subset)

West Ruislip	Ruislip Gardens	South Ruislip	Northolt
Greenford	Perivale	Hanger Lane	Ealing Broadway
West Acton	North Acton	East Acton	White City
Shepherd's Bush	Holland Park	Notting Hill Gate	Queensway
Lancaster Gate	Marble Arch	Bond Street	Oxford Circus
Tottenham Court Road	Holborn	Chancery Lane	St. Paul's
Bank	Liverpool Street	Bethnal Green	Mile End
Stratford	Leyton	Leytonstone	Wanstead
Redbridge	Gants Hill	Newbury Park	Barkingside
Fairlop	Hainault	Grange Hill	Chigwell
Roding Valley	Snaresbrook	South Woodford	Woodford
Buckhurst Hill	Loughton	Debden	Theydon Bois
Epping			

Regular Expression Commands

There are six regular expression functions within Db2 including:

- REGEXP_COUNT – Returns a count of the number of times that a regular expression pattern is matched in a string.
- REGEXP_EXTRACT – Returns one occurrence of a substring of a string that matches the regular expression pattern.
- REGEXP_INSTR – Returns the starting or ending position of the matched substring, depending on the value of the return option argument.
- REGEXP_LIKE – Returns a Boolean value indicating if the regular expression pattern is found in a string. The function can be used only where a predicate is supported.
- REGEXP_MATCH_COUNT – Returns a count of the number of times that a regular expression pattern is matched in a string.
- REGEXP_REPLACE – Returns a modified version of the source string where occurrences of the regular expression pattern found in the source string are replaced with the specified replacement string.
- REGEXP_SUBSTR – Returns one occurrence of a substring of a string that matches the regular expression pattern.

Each one of these functions follows a similar calling sequence:

```
REGEXP_FUNCTION(source, pattern, flags, start_pos, codeunits)
```

The arguments to the function are:

- Source – string to be searched
- Pattern – the regular expression that contains what we are searching for
- Flag – settings that control how matching is done
- Start_pos – where to start in the string
- Codeunits – which type unit of measurement start_pos refers to (for Unicode)

The source can be any valid Db2 string including CHAR, VARCHAR, CLOB, etc. Start_pos is the location in the source string that you want to start searching from, and codeunits tells Db2 whether to treat the start_pos as an absolute location (think byte location) or a character location which takes into account the Unicode size of the character string.

Codeunits can be specified as CODEUNITS16, CODEUNITS32, or OCTETS. CODEUNITS16 specifies that start is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that start is expressed in 32-bit UTF-32 code units. OCTETS specify that start is expressed in bytes.

Pattern and flag values are complex and so are discussed in the following sections.

Regular Expression Flag Values

Regular expression functions have a flag specification that can be used to change the behavior of the search. There are six possible flags that can be specified as part of the REGEXP command:

Table 11: Regular Expression Flags

Flag	Meaning
c	Specifies that matching is case-sensitive (the default value)
I	Specifies that matching is case insensitive
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

Regular Expression Search Patterns

Regular expressions use certain characters to represent what is matched in a string. The simplest pattern is a string by itself.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'Ruislip');

STATION
--------------------
West Ruislip
Ruislip Gardens
South Ruislip
```

The pattern 'Ruislip' will look for a match of Ruislip within the STATION column. Note that this pattern will also match 'West Ruislip' or 'Ruislip Gardens' since we placed no restriction on where the pattern can be found in the string. The match will also be exact (case matters). This type of search would be equivalent to using the SQL LIKE statement:

```
SELECT STATION FROM CENTRAL_LINE
  WHERE STATION LIKE '%Ruislip%';

STATION
--------------------
West Ruislip
Ruislip Gardens
South Ruislip
```

If you didn't place the % at the beginning of the LIKE string, only the stations that start with Ruislip would be found.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE STATION LIKE 'Ruislip%';

STATION
--------------------
Ruislip Gardens
```

If you want to match Ruislip with upper or lower case being ignored, you would add the 'i' flag as part of the REGEXP_LIKE (or any REGEXP function).

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'RUISLIP','i');

STATION
--------------------
West Ruislip
Ruislip Gardens
South Ruislip
```

Anchoring Patterns in a Search

By default, a pattern will be matched anywhere in a string. Our previous example showed how Ruislip could be found anywhere in a string. To force a match to start at the beginning of a string, the carat symbol ^ can be used to force the match to occur at the beginning of a string.

```
 SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'^Ruislip');

STATION
--------------------
Ruislip Gardens
```

To match a pattern at the end of the string, the dollar sign $ can be used.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'Ruislip$');

STATION
--------------------
West Ruislip
South Ruislip
```

To force an exact match with a string you would use both the beginning and end anchors.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'^Leyton$');

STATION
--------------------
Leyton
```

Note that if we didn't use the end anchor, we are going to get more than one result.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'^Leyton');

STATION
--------------------
Leyton
Leytonstone
```

Matching patterns across multiple lines

So far the examples have dealt with strings that do not contain newline characters (or carriage feeds). In some applications, data from an input panel may include multiple lines which may contain hard line feeds. What this means is that there are multiple lines in the data, but from a database perspective, there is only one line in the VARCHAR field. You can modify the behavior of the Regular Expression search by instructing it to honor the CRLF characters as line delimiters.

The following SQL will insert a single line with multiple CRLF characters in it to simulate a multi-line text string.

```
CREATE TABLE LONGLINE (NAME VARCHAR(255));

INSERT INTO LONGLINE
  VALUES 'George' || CHR(10) || 'Katrina';
```

Searching for Katrina at the beginning and end of string doesn't work.

```
SELECT COUNT(*) FROM LONGLINE
  WHERE REGEXP_LIKE(NAME,'^Katrina$');

1
-----------
          0
```

We can override the regular expression search by telling it to treat each NL/CRLF as the end of a string within a string.

```
SELECT COUNT(*) FROM LONGLINE
  WHERE REGEXP_LIKE(NAME,'^Katrina$','m');

1
-----------
          1
```

Logical OR Operator

Regular expressions can match more than one pattern. The OR operator (|) is used to define alternative patterns that can match in a string. The following example searches for stations that have "ing" in their name as well as "hill".

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'way|ing');

STATION
--------------------
Ealing Broadway
Notting Hill Gate
Queensway
Barkingside
Roding Valley
Epping
```

Some things to be aware of when creating the search pattern. Spaces in the patterns themselves are significant. If the previous search pattern had a space in one of the words, it would not find it (unless of course there was a space in the station name).

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'way| ing');

STATION
--------------------
Ealing Broadway
Queensway
```

Using the "x" flag will ignore blanks in your pattern, so this would fix issues that we have in the previous example.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'way| ing','x');

STATION
--------------------
Ealing Broadway
Notting Hill Gate
Queensway
Barkingside
Roding Valley
Epping
```

Brackets can be used to make it clear what the pattern is that you are searching for and avoid the problem of having blanks in the expression. Brackets do have a specific usage in regular expressions, but here we are using it only to separate the two search strings.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(way)|(ing)');

STATION
--------------------
Ealing Broadway
Notting Hill Gate
Queensway
Barkingside
Roding Valley
Epping
```

Combining Patterns

As we found out in the previous section, there is an OR operator that you can use to select between two patterns. How do you request that multiple patterns be present? First, we must understand how matching occurs when we have multiple strings that need to be matched that have an unknown number of characters between them.

For instance, how do we create a pattern that looks for "ing" followed by "way" somewhere in the string? Regular expressions recognize the "." (period) character as matching anything. Following the pattern, you can add a modifier that specifies how many times you want the pattern matched:

Table 12: London Central Line (Subset)

Char	Meaning
*	Match zero or more times
?	Match zero or one times
+	Match one or more times
{m}	Match exactly m times
{m,}	Match as least a minimum of m times
{m,n}	Match at least a minimum of m times and no more than n times

The following regular expression searches for a pattern with "ing" followed by any characters and then "way".

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(ing)*.(way)');

STATION
--------------------
Ealing Broadway
Queensway
```

The previous answer gave you two results (Ealing Broadway and Queensway). Why two? The reason is that we used the * in the wrong place (a single character in a wrong place can result in very different results!). What we really needed to do was place a .* after the (ing) to match "ing" and then any characters, before matching "way". What our query did above was match 0 or more occurrences of "ing", followed by any character, and then match "way". Here is the correct query.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(ing).*(way)')

STATION
-------------------
Ealing Broadway
```

Finding at least one occurrence of a pattern requires the use of the + operator, or the bracket operators. This example locates at least one occurrence of the "an" string in station names.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(an)+');

STATION
--------------------
Hanger Lane
Holland Park
Lancaster Gate
Chancery Lane
Bank
Wanstead
Gants Hill
Grange Hill
```

If we want to find an exact number of occurrences, we need to use the {} notation to tell the regular expression matcher how many we want to find. The syntax of the {} match is:

- {m} – Match exactly m times
- {m,} – Match as least a minimum of m times
- {m,n} – Match at least a minimum of m times and no more than n times

So the "+" symbol is equivalent to the following regular expression using the {} syntax.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(an){1,}');

STATION
--------------------
Hanger Lane
Holland Park
Lancaster Gate
Chancery Lane
Bank
Wanstead
Gants Hill
Grange Hill
```

If we want to match exactly 2 'an' patterns in a string, we would think that changing the expression to {2} would work.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(an){2}');

No records found.
```

Sadly, we get no results! This would appear to be the wrong result, but it's because we got lucky with our first search! The best way to figure out what we matched in the original query is to use the REGEXP_EXTRACT and REGEXP_INSTR functions.

- REGEXP_EXTRACT – Returns one occurrence of a substring of a string that matches the regular expression pattern.
- REGEXP_INSTR – Returns the starting or ending position of the matched substring, depending on the value of the return option argument.

The following SQL gives us a clue to what was found with the (an) pattern.

```
SELECT STATION,
   REGEXP_INSTR(STATION,'(an)') AS LOCATION,
   REGEXP_EXTRACT(STATION,'(an)') AS EXTRACT
FROM CENTRAL_LINE
   WHERE REGEXP_LIKE(STATION,'(an)');

STATION              LOCATION    EXTRACT
-------------------- ----------- --------------------
Hanger Lane                    2 an
Holland Park                   5 an
Lancaster Gate                 2 an
Chancery Lane                  3 an
Bank                           2 an
Wanstead                       2 an
Gants Hill                     2 an
Grange Hill                    3 an
```

What you should see in the previous result is the location where the "an" pattern was found in the string. Note that all we matched was the "an" pattern, nothing else. So why can't I find two "an" patterns in the string? The reason is that (an){2} means "an" followed by another "an"! We didn't tell the pattern to match anything else! What we need to do is modify the pattern to say that it can match "an" followed by anything else. The pattern needs to be modified to (an).* where the ".*" means any character following the "an".

In order to tell the regular expression function to use this entire pattern (an).* twice, we need to place brackets around it as well. The final pattern is ((an).*){2}.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'((an).*){2}');

STATION
--------------------
Hanger Lane
Chancery Lane
```

You should find that two stations match the pattern. The following SQL shows which pattern is matched first in the STATIONS names.

```
SELECT STATION,
   REGEXP_INSTR(STATION,'((an).*){2}') AS LOCATION,
   REGEXP_EXTRACT(STATION,'((an).*){2}') AS EXTRACT
FROM CENTRAL_LINE
   WHERE REGEXP_LIKE(STATION,'((an).*){2}');

STATION              LOCATION    EXTRACT
-------------------- ----------- --------------------
Hanger Lane                    2 anger Lane
Chancery Lane                  3 ancery Lane
```

Matching character types

Aside from matching entire strings, you can also use regular expression to look for patterns of characters. The simplest matching pattern is the period (.) which matches any character. Matching a string of arbitrary length is the pattern ".*". The "+" and "?" characters can also be used to modify how many characters you want matched.

What about situations where you want to check for certain patterns or characters in a string? A good example would be a social security number, or credit card number. There are certain patterns that you would find for these objects. Assume we have a social security number in the format xxx-xx-xxxx. It is possible to create a regular expression that would return true if the SSN matched the pattern above (it doesn't tell you if the SSN itself is valid, only that it has the proper format).

Regular expressions allow you to create a list of characters that need to be matched in something called a bracket expression. A bracket expression has the format:

```
[a-z] [A-Z] [0-9] [a-zA-z]
```

The examples above represent the following search patterns:

- [a-z] - match any series of lowercase characters between a and z
- [A-Z] - match any series of uppercase characters between A and Z
- [0-9] - match any valid digits
- [a-zA-Z] - match any lower- or uppercase letters

You can also enumerate all of the characters you want to match by listing them between the brackets like [abcdefghikjlmnopqrstuvwxyz]. The short form a-z is easier to read and less prone to typing errors!

The following example checks for station names that begin with the letter P-R.

```
SELECT STATION FROM CENTRAL_LINE
   WHERE REGEXP_LIKE(STATION,'^[P-R]');

STATION
--------------------
Ruislip Gardens
Perivale
Queensway
Redbridge
Roding Valley
```

If you wanted to include all stations that have the letter P-R or p-e, you could add the condition within the brackets.

```
SELECT STATION FROM CENTRAL_LINE
   WHERE REGEXP_LIKE(STATION,'[p-rP-R]');

STATION
--------------------
West Ruislip
Ruislip Gardens
South Ruislip
Northolt
Greenford
Perivale
Hanger Lane
Ealing Broadway
North Acton
Shepherd's Bush
```

Back to our SSN question. Can a regular expression pattern be used to determine whether the string is in the correct format? The data will be in the format XXX-XX-XXXX so the regular expression needs to find the three numeric values separated by dashes.

The number pattern can be represented with the bracket expression [0-9]. To specify the number of characters that need to be found, we use the braces {} to specify the exact number required.

For the three numbers in the pattern we can use [0-9]{3}, [0-9]{2}, and [0-9]{4}. Adding in the dashes gives us the final pattern. The SQL below checks to see if a SSN is correct.

```
VALUES
  CASE
    WHEN REGEXP_LIKE('123-34-1422','[0-9]{3}-[0-9]{2}-[0-9]{4}')
      THEN 'Valid'
    ELSE 'Invalid'
  END;

1
-------
Valid
```

The SSN is valid in the example above. Here are some other examples to show whether or not the regular expression picks up all of the errors.

```
WITH SSNS(SSN) AS (
   VALUES
      '123-34-1322',
      'ABC-34-9999',
      'X123-44-0001',
      '123X-Y44-Z0001',
      '111-222-111'
  )
SELECT SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'[0-9]{3}-[0-9]{2}-[0-9]{4}')
      THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS;

SSN            2
-------------- -------
123-34-1322    Valid
ABC-34-9999    Invalid
X123-44-0001   Valid
123X-Y44-Z0001 Invalid
111-222-111    Invalid
```

If you check closely, one of the strings was marked as valid, although it is not correct (X123-44-0001). The reason this occurred is that the pattern was found after the "X" and it was correct. To prevent this from happening, we need to anchor the pattern at the beginning to avoid this situation. A better pattern would be to anchor both ends of the pattern so there is no possibility of other characters being at the beginning or end of the pattern.

```
WITH SSNS(SSN) AS (
   VALUES
      '123-34-1322',
      'ABC-34-9999',
      'X123-44-0001',
      '123X-Y44-Z0001',
      '111-222-111'
  )
SELECT SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'^[0-9]{3}-[0-9]{2}-[0-9]{4}$')
      THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS;

SSN            2
-------------- -------
123-34-1322    Valid
ABC-34-9999    Invalid
X123-44-0001   Invalid
123X-Y44-Z0001 Invalid
111-222-111    Invalid
```

Special Patterns

The previous example used the [0-9] syntax to request that only numbers be found in the pattern. There are some predefined patterns that define these common patterns. The first argument is Posix format (if it exists), the second is the escape character equivalent, and the final one is the raw pattern it represents.

Table 13: Regular Expression Special Patterns

Posix	Escape	Pattern	Meaning
[:alnum:]		[A-Za-z0-9]	Alphanumeric characters
	\w	[A-Za-z0-9_]	Alphanumeric characters plus "_"
	\W	[^A-Za-z0-9_]	Non-word characters
[:alpha:]	\a	[A-Za-z]	Alphabetic characters
[:blank:]	\s, \t		Space and tab
	\b		Word boundaries
[:cntrl:]		[\x00-\x1F\x7F]	Control characters
[:digit:]	\d	[0-9]	Digits
	\D	[^0-9]	Non-digits
[:graph:]		[\x21-\x7E]	Visible characters
[:lower:]	\l	[a-z]	Lowercase letters
[:print:]	\p	[\x20-\x7E]	Visible characters and the space character

[:punct:]		[][!"#$%& '()*+,./:;<=>?@\^_`{ |}~-]	Punctuation characters
[:space:]	\s	[\t\r\n\v\f]	Whitespace characters
	\S	[^ \t\r\n\v\f]	Non-whitespace characters
[:upper:]	\u	[A-Z]	Uppercase letters
[:xdigit:]	\x	[A-Fa-f0-9]	Hexadecimal digits

For instance, the following three statements will produce the same result.

```
WITH SSNS(SSN) AS (
   VALUES
      '123-34-1322',
      'ABC-34-9999',
      'X123-44-0001',
      '123X-Y44-Z0001',
      '111-222-111'
  )
SELECT 'Original', SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'^[0-9]{3}-[0-9]{2}-[0-9]{4}$') THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS
UNION ALL
SELECT 'Posix', SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'^[:digit:]{3}-[:digit:]{2}-[:digit:]{4}$')
      THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS
UNION ALL
SELECT 'Escape', SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'^\d{3}-\d{2}-\d{4}$') THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS;
```

```
1        SSN            3
-------- -------------- -------
Original 123-34-1322    Valid
Original ABC-34-9999    Invalid
Original X123-44-0001   Invalid
Original 123X-Y44-Z0001 Invalid
Original 111-222-111    Invalid
Posix    123-34-1322    Valid
Posix    ABC-34-9999    Invalid
Posix    X123-44-0001   Invalid
Posix    123X-Y44-Z0001 Invalid
Posix    111-222-111    Invalid
```

Negating Patterns

Up to this point in time, the patterns that have been used are looking for a positive match. In some cases, you may want to find values that do not match. The easiest way is to negate the actual REGEXP_LIKE expression. The following expression finds all of the stations that start and end with "West".

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'^West$');

No records found.
```

Adding the NOT modifier in front of the REGEXP function gives us the stations that do not begin with West.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE NOT REGEXP_LIKE(STATION,'^West$');

STATION
--------------------
West Ruislip
Ruislip Gardens
South Ruislip
Northolt
Greenford
Perivale
Hanger Lane
Ealing Broadway
West Acton
North Acton
```

You can also negate some of the searches in a pattern by using the [^...] syntax where the ^ tells the regular expression not to match the following characters. The expression [^0-9] would mean match any characters which are not numeric.

However, regular expressions have something called negative lookarounds which basically mean find the pattern which does not match. You create this pattern by adding the (?!..) at the beginning of the string. The same query (finding stations that don't start with West) would be written with this lookaround logic found in the SQL below.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'^(?!West)');

STATION
--------------------
Ruislip Gardens
South Ruislip
Northolt
Greenford
Perivale
Hanger Lane
Ealing Broadway
```

```
North Acton
East Acton
White City
```

Capturing Parenthesis

The previous example used something called a negative lookaround with capturing parenthesis. When you place a pattern within a set of brackets (...) the string that matches this pattern is "remembered". The strings that are matched can be used in subsequent parts of your regular expression. This allows a form of programming within your regular expression!

Each set of parentheses that are matched are associated with a number, starting at one and incrementing for each subsequent pattern match. For instance, the following pattern will have three matches:

```
^([0-9]{3})-([0-9]{3})-([0-9]{3})$
```

This is like the SSN example used earlier on in this section. The difference in this example is that each block of numbers is the same (3 digits). This pattern will match any sequence of numbers in the format 123-456-789.

```
WITH SSNS(SSN) AS (
   VALUES
      '123-456-789',
      '123-555-123',
      '890-533-098',
      '123-456-456'
  )
SELECT SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'^([0-9]{3})-([0-9]{3})-([0-9]{3})$')
      THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS;

SSN         2
----------- -------
123-456-789 Valid
123-555-123 Valid
890-533-098 Valid
123-456-456 Valid
```

These numbers fit the pattern and should be valid. When one of the capturing parenthesis matches, it will remember the string that it matched. For instance, in the first example (123-456-789), the first match will find the string '123'. The second match will find '456' and so on. We can refer to these matched strings with the special control characters \n where n represents the capturing parenthesis location. So \1 would refer to the '123' that was found. \2 would be for '456'.

The regular expression will be updated so that the last part of the pattern needs to be numeric (0-9) but can't be same as the first match.

```
WITH SSNS(SSN) AS (
   VALUES
      '123-456-789',
      '123-555-123',
      '890-533-098',
      '123-456-456'
  )
SELECT SSN,
  CASE
    WHEN REGEXP_LIKE(SSN,'^([0-9]{3})-([0-9]{3})-(?!\1)([0-9]{3})$')
      THEN 'Valid'
    ELSE 'Invalid'
  END
FROM SSNS;
```

```
SSN         2
----------- -------
123-456-789 Valid
123-555-123 Invalid
890-533-098 Valid
123-456-456 Valid
```

In many cases, it may be easier to find the patterns that match and then negate the REGEXP statement!

The (?...) syntax is used for a variety of purposes in regular expressions:

Table 14: Capturing Parentheses

Pattern	Description	Details
(?: ...)	Non-capturing parentheses	Groups the included pattern, but does not provide capturing of matching text. More efficient than capturing parentheses.
(?> ...)	Atomic-match parentheses	First match of the parenthesized subexpression is the only one tried. If it does not lead to an overall pattern match, back up the search for a match to a position before the "(?>"
(?# ...)	Free-format comment	(?# comment)
(?= ...)	Look-ahead assertion	True if the parenthesized pattern matches at the current input position, but does not advance the input position.
(?! ...)	Negative look-ahead assertion	True if the parenthesized pattern does not match at the current input position. Does not advance the input position.

(?<= ...)	Look-behind assertion	True if the parenthesized pattern matches text that precedes the current input position. The last character of the match is the input character just before the current position. Does not alter the input position. The length of possible strings that is matched by the look-behind pattern must not be unbounded (no * or + operators.)
(?<!...)	Negative Look-behind assertion	True if the parenthesized pattern does not match text that precedes preceding the current input position. The last character of the match is the input character just before the current position. Does not alter the input position. The length of possible strings that is matched by the look-behind pattern must not be unbounded (no * or + operators.)

For efficiency in matching, the best approach is to place strings that you are searching for in non-capturing parentheses (?:...) rather than the generic () parenthesis. The following example finds all stations with "West" in the name.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(West)');

STATION
--------------------
West Ruislip
West Acton
```

The following SQL is equivalent, except that the matched pattern is not kept for future use in matching.

```
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'(?:West)');

STATION
--------------------
West Ruislip
West Acton
```

10

Statistical Functions

NEW AND UPDATED STATISTICAL FUNCTIONS FOR COMPLEX DATA ANALYSIS

Statistical Functions

Db2 already has a variety of Statistical functions built in. In Db2 11.1, a number of new functions have been added including:

- COVARIANCE_SAMP – Sample covariance of a set of number pairs
- STDDEV_SAMP – Sample standard deviation
- VARIANCE_SAMP – Sample variance of a set of numbers
- CUME_DIST - Cumulative Distribution
- PERCENT_RANK - Percentile rank
- PERCENTILE_DISC and PERCENTILE_CONT – Percentiles
- MEDIAN
- WIDTH_BUCKET

Sampling Functions

The traditional VARIANCE, COVARIANCE, and STDDEV functions have been available in Db2 for a long time. When computing these values, the formulae assume that the entire population has been counted (N). The traditional formula for standard deviation is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

N refers to the size of the population and in many cases, we only have a sample, not the entire population of values.

In this case, the formula needs to be adjusted to account for the sampling.

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

The three functions that use a sampling algorithm are COVARIANCE_SAMP, STDDEV_SAMP, and VARIANCE_SAMP.

COVARIANCE_SAMP returns the sample covariance of a set of number pairs.

```
SELECT COVARIANCE_SAMP(SALARY, BONUS)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';

1
------------------------------------------
                                   5428750
```

STDDEV_SAMP returns the sample standard deviation (division by [n-1]) of a set of numbers.

```
SELECT STDDEV_SAMP(SALARY)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';

1
------------------------------------------
       46863.03180546474203969595277486485
```

VARIANCE_SAMP returns the sample variance (division by [n-1]) of a set of numbers.

```
SELECT VARIANCE_SAMP(SALARY)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';

1
------------------------------------------
                                2196143750
```

Additional Analytical Functions

Db2 11.1 also includes several analytic functions that Data Scientists will find useful. These functions include MEDIAN, CUME_DIST, PERCENT_RANK, PERCENTILE_DISC, PERCENTILE_CONT, and WIDTH_BUCKET.

The MEDIAN column function returns the median value in a set of values.

```
SELECT MEDIAN(SALARY) AS MEDIAN, AVG(SALARY) AS AVERAGE
  FROM EMPLOYEE WHERE WORKDEPT = 'E21';


MEDIAN                          AVERAGE
----------------------------- ------------------------------------
                        41895  47086.6666666666666666666666666667
```

CUME_DIST returns the cumulative distribution of a row that is hypothetically inserted into a group of rows.

```
SELECT CUME_DIST(47000) WITHIN GROUP (ORDER BY SALARY)
  FROM EMPLOYEE WHERE WORKDEPT = 'A00';

1
------------------------------------------
                                       0.5
```

The PERCENT_RANK column function returns the relative percentile rank of a row that is hypothetically inserted into a group of rows.

```
SELECT PERCENT_RANK(47000) WITHIN GROUP (ORDER BY SALARY)
  FROM EMPLOYEE WHERE WORKDEPT = 'A00';

1
------------------------------------------
                                       0.4
```

The PERCENTILE_DISC returns the value that corresponds to the specified percentile given a sort specification by using discrete (DISC) or continuous (CONT) distribution.

```
SELECT PERCENTILE_DISC(0.75) WITHIN GROUP (ORDER BY SALARY)
  FROM EMPLOYEE WHERE WORKDEPT = 'E21';

1
-----------
   45370.00

SELECT PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY SALARY)
  FROM EMPLOYEE WHERE WORKDEPT = 'E21';

1
----------------------
 4.49875000000000E+004
```

WIDTH BUCKET and Histogram Example

The WIDTH_BUCKET function is used to create equal-width histograms. Using the EMPLOYEE table, this SQL will assign a bucket to each employee's salary using a range of 35000 to 100000 divided into 13 buckets.

```
SELECT EMPNO, SALARY, WIDTH_BUCKET(SALARY, 35000, 100000, 13)
  FROM EMPLOYEE ORDER BY EMPNO;

EMPNO  SALARY      3
------ ----------- -----------
000010   152750.00          14
000020    94250.00          12
000030    98250.00          13
000050    80175.00          10
000060    72250.00           8
000070    96170.00          13
000090    89750.00          11
000100    86150.00          11
000110    66500.00           7
000120    49250.00           3
```

We can plot this information by adding some more details to the bucket output.

```
WITH BUCKETS(EMPNO, SALARY, BNO) AS (SELECT EMPNO, SALARY,
   WIDTH_BUCKET(SALARY, 35000, 100000, 13) aS BUCKET
FROM EMPLOYEE ORDER BY EMPNO)
SELECT BNO, COUNT(*) FROM BUCKETS
GROUP BY BNO
ORDER BY BNO ASC;

BNO         2
----------- -----------
          0           1
          1          10
          2           3
          3           8
          4           2
          5           2
          6           2
          7           5
          8           2
         10           1
```

Plotting the data results in the following bar chart.

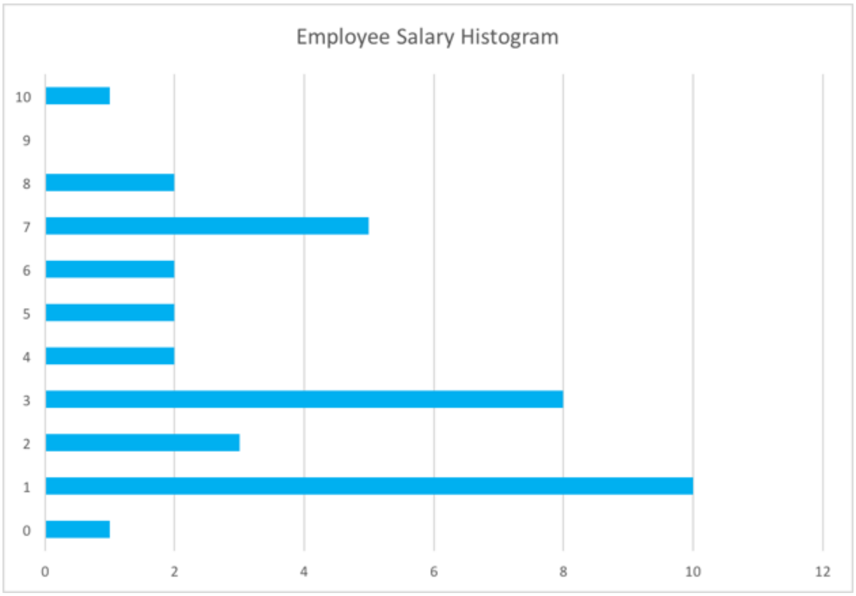


Figure 16: Histogram of Employee Salaries

11

Hashing Functions

Hashing Functions for Generating Unique Values for Character Strings

Hashing Functions

Hashing functions are typically used to take larger text strings and map into a smaller "hashed" value. These hash values can be 4 byte (HASH4), 8 byte (HASH8) or up to 64 bytes long (HASH). The reason for using various HASH algorithms depends on the size of the data and how many collisions you expect to get. A HASH function does not necessarily result in a unique value for every string that is hashed. To avoid collisions, a large HASH value is often used. The tradeoff is the amount of compute power required to generate the HASH value as well as the size of the HASH key.

HASH values can be used for a variety of purposes. HASH values can be used to create artificial partition keys for a table, create a checksum for transporting the data to an application or another database, or for indexing phrases. The application must be aware that duplicates could occur so additional coding may be required.

HASH 4 - Four-byte Hash Encoding

HASH4 is the simplest of the hashing functions, but also has the potential for the most collisions when computing hash values. The HASH4 takes two arguments: the string to be hashed, and the algorithm that is to be used for the hash.

```
HASH4(string, algorithm)
```

The HASH4 function returns a 4-byte integer value. The algorithm value can be either 0 or 1. Zero (0) is the default value and uses the Adler algorithm. One (1) uses the CRC32 algorithm. The Adler algorithm provides a faster checksum hash; however, it has poor coverage when the strings are less than a few hundred bytes in size. Poor coverage means that two different strings hash to the same value, referred to as a "collision". In this case, use the CRC32 algorithm, or switch to HASH8 instead.

The following SQL will HASH a phrase using the default (Adler) algorithm.

```
VALUES HASH4('Hello there');

1
-----------
  409338925
```

Changing the phrase slightly results in a different HASH value.

```
VALUES HASH4('Hello therf');

1
-----------
  409404462
```

The SQL is modified to use the CRC32 algorithm.

```
VALUES HASH4('Hello there',1);

1
-----------
 -342989177
```

HASH 8 - Eight-byte Hash Encoding

The HASH8 function is identical to the HASH4 function except that it returns an 8-byte (BIGINT) value. There are two arguments to the HASH8 function. The first is the string to be hashed and the second is the algorithm to use. With this version of Db2, there is only one value allowed (0) which tells the function to use the Jenkins algorithm.

```
   HASH8(string, algorithm)
```

The HASH8 function will have a higher probability of returning unique values for all strings, but the downside is that it will require more cycles to compute the hash value.

```
VALUES HASH8('Hello there');

1
--------------------
 -844407019926684877
```

Supplying anything other than a zero for the second argument will result in an error being returned.

```
VALUE HASH8('Hello there',1);

SQL0104N  An unexpected token "VALUE HASH8" was found following
"BEGIN-OF-STATEMENT". Expected tokens may include:
"<create_index>". SQLSTATE=42601
```

HASH - Variable Hash Encoding

The HASH function is the most flexible of all the hashing functions. It has a format identical to the other two HASH functions:

```
   HASH(string, algorithm)
```

There are four choices for algorithm:

Table 15: HASH Algorithm Options

#	Algorithm	Length	Bytes	Return Values
0	MD5	128	16	2**128
1	SHA1	160	20	2**160
2	SHA2_256	256	32	2**256
3	SHA2_512	512	64	2**512

The result of the function is a VARBINARY field. Note that the MD5 and SHA1 algorithms have had security flaws identified with them so you may want to consider using a higher strength algorithm if you are using this for security purposes.

```
VALUES HASH('Hello there',0),
       HASH('Hello there',1),
       HASH('Hello there',2),
       HASH('Hello there',3);

1
-----------------------------------------------------------------
E8EA7A8D1E93E8764A84A0F3DF4644DE
726C76553E1A3FDEA29134F36E6AF2EA05EC5CCE
4E47826698BB4630FB4451010062FADBF85D61427CBDFAED7AD0F23F239BED89
567683DDBA1F5A576B68EC26F41FFBCC7E718D646839AC6C2EF746FE952CEF4CBE6DEA...
```

12

JSON Functions

Extending Db2 to Manipulate JSON Documents within Relational Tables

Internal Db2 JSON Routines

The current Db2 11.1 release includes a driver-based JSON solution that embraces the flexibility of the JSON data representation within a relational database. Users can use a JSON programming paradigm that is modeled after the MongoDB data model and query language and store the data natively in Db2. The JSON data is stored in a binary-encoded format called BSON (Binary JSON). BSON is designed to be lightweight, easily traversed and very efficiently encoded and decoded.

Users can interact with JSON data in many ways.

- They can administer and interactively query JSON data using a command line shell.
- They can programmatically store and query data from Java programs using an IBM provided Java driver for JSON that allows users to connect to their JSON data through the same JDBC driver used for SQL access.
- They can use any driver that implements portions of the MongoDB protocol. This allows them to access their Db2 JSON store from a variety of modern languages, including node.js, PHP, Python, and Ruby, as well as more traditional languages such as C, C++, and Perl.

These approaches to manipulating JSON within Db2 use a variety of user-defined functions (UDFs). These UDFs are catalogued whenever a JSON data store is defined within Db2. Developers do not have access to these SQL-based routines since they were originally designed to be used only by these specific JSON interfaces. However, many customers would like to store and query a JSON column that is defined as part of a Db2 table. While these routines have not been officially published, they are available for customer use.

These routines are not externalized in the Db2 documentation because they were only used by the internal API's for querying and managing JSON. For this reason, these routines were not originally considered "supported". This chapter gives you details of these functions and they are now available to use in your applications. Note that the internal use of these routines by Db2 has very specific usage patterns, which means that it is possible that you may generate a set of SQL that may not be handled properly.

Note: These functions may change in the future to conform to the SQL standard.

Db2 JSON Functions

There is one built-in Db2 JSON function and several other functions that must be registered within Db2 before they can be used. The names of the functions and their purpose are described below.

- JSON_VAL – Extracts data from a JSON document into SQL data types
- JSON_TABLE – Returns a table of values for a document that has array types in it
- JSON_TYPE – Returns the data type of a specific field within a JSON document
- JSON_LEN – Returns the count of elements in an array type inside a document
- BSON2JSON – Convert BSON formatted document into JSON strings
- JSON2BSON – Convert JSON strings into a BSON document format
- JSON_UPDATE – Update a field or document using set syntax
- JSON_GET_POS_ARR_INDEX – Find a value within an array
- BSON_VALIDATE – Checks to make sure that a BSON field in a BLOB object is in a correct format

Aside from the JSON_VAL function, all other functions in this list must be catalogued before first being used. A script that can be used to catalog these functions (DB2-V11-JSON-Examples.sql) is available in the download folder for this eBook. In addition, all the examples in this chapter are also included in this script. Please refer to the appendix for details on accessing this file and other useful information.

❷Path Statement Requirements

Three of the Db2 JSON functions are catalogued in the SYSIBM schema (JSON_VAL, JSON2BSON, BSON2JSON) while the remainder are catalogued under the SYSTOOLS schema. Prior to FP2, all of the JSON functions needed to be catalogued manually. As of FP2, the cataloguing is done automatically for you when you upgrade the instance or the initial installation.

If you are at a prior release of Db2 (11.1.1.1 and 10.5) all the Db2 JSON functions have been placed into the SYSTOOLS schema. This means that to execute any of these commands, you must prefix the command with SYSTOOLS, as in SYSTOOLS.JSON2BSON. To remove this requirement, you must update the CURRENT PATH value to include SYSTOOLS as part of it. The SQL below will tell you what the current PATH is.

```
VALUES CURRENT PATH;

1
-------------------------------------------------
"SYSIBM","SYSFUN","SYSPROC","SYSIBMADM","BAKLARZ"
```

Use the following SQL to add SYSTOOLS to the current path.

```
SET CURRENT PATH = CURRENT PATH, SYSTOOLS;
```

From this point on you won't need to add the SYSTOOLS schema on the front of any of your SQL statements that refer to these Db2 JSON functions.

Creating Tables that Support JSON Documents

To create a table that will store JSON data, you need to define a column that can contain binary data. The JSON column **must** be created as a BLOB (binary object) data type because the new VARBINARY data type will not work with any of the JSON functions. JSON Documents are always stored in BLOBS which can be as large as 16M.

To ensure good performance, you should have a BLOB specified as INLINE if possible. If a large object is not inlined, or greater than 32K in size, the resulting object will be placed into a large table space. The result is that BLOB objects will not be kept in bufferpools (which means a direct read is required from disk for access to any BLOB object) and that two I/Os are required to get any document – one for the row and a second for the document. By using the INLINE option and keeping the BLOB size below the page size, you can improve the retrieval performance of JSON columns.

This SQL will create a column that is suitable for storing JSON data:

```
CREATE TABLE TESTJSON
  (
  JSON_FIELD BLOB(4000) INLINE LENGTH 4000
  );
```

Note that we are assuming that the size of the JSON object will not exceed 4000 characters in size.

JSON Document Representation

Db2 stores JSON objects in a modified binary-encoded format called BSON (Binary JSON). As mentioned at the beginning of this chapter, BSON is designed to be lightweight, easily traversed and very efficiently encoded and decoded. All the JSON functions execute against BSON objects, not the original JSON text strings that may have been generated by an application.

To store and retrieve an entire document from a column in a table, you must use:

- BSON2JSON – Convert a BSON formatted document into a JSON string
- JSON2BSON – Convert JSON strings into a BSON document format

You can also verify the contents of a document that is stored in a column by using the BSON_VALIDATE function:

- BSON_VALIDATE – Checks to make sure that a BSON field in a BLOB object is in the correct format

You cannot use JSON to BSON conversion functions that are available in some programming languages. The BSON format used by Db2 has some extensions to it that will not be handled by these routines, so you must use the Db2 BSON2JSON and JSON2BSON functions exclusively.

JSON2BSON: Inserting a JSON Document

Inserting into a Db2 column requires the use of the JSON2BSON function. The JSON2BSON function (and BSON2JSON) are used to transfer data in and out of a traditional Db2 BLOB column. Input to the JSON2BSON function must be a properly formatted JSON document. If the document does not follow proper JSON rules, you will get an error code from the function.

```
INSERT INTO TESTJSON VALUES
  ( JSON2BSON('{Name:"George"}'));
```

A poorly formatted JSON document will return an error.

```
INSERT INTO TESTJSON VALUES
  ( JSON2BSON('{Name:, Age: 32}'));

SQL0443N  Routine "JSON2BSON" (specific name "") has returned an
error SQLSTATE with diagnostic text "JSON parsing error for:
{Name:, Age: 32}, error code: 4 ".  SQLSTATE=22546
```

BSON2JSON: Retrieving a JSON Document

Since the data that is stored in a JSON column is in a special binary format called BSON, selecting from the field will only result in random characters being displayed.

```
SELECT * FROM TESTJSON;

JSON_FIELD
--------------------
03160000000024E616D65
```

If you want to extract the contents of a JSON field, you must use the BSON2JSON function.

```
SELECT BSON2JSON(JSON_FIELD) FROM TESTJSON;

1
--------------------
{"Name":"George"}
```

One thing that you will notice is that the retrieved JSON has been modified slightly so that all the values have quotes around them to avoid any ambiguity. This is due to the conversion to BSON format and back to JSON. Note that we didn't necessarily require the quotes when we inserted the data. For instance, our original JSON document contained the following field:

```
{Name:"George"}
```

What gets returned is slightly different, but still considered to be the same JSON document.

```
{"Name":"George"}
```

You must ensure that the naming of any fields is consistent between documents. "Name", "name", and "Name" are all considered different fields. One option is to use lowercase field names, or to use camel-case (first letter is capitalized) in all your field definitions. The important thing is to keep the naming consistent so you can find the fields in the document.

Determining Document Size

The size of BLOB field will be dependent on the size of documents that you expect to store in them. Converting a JSON document into a BSON format usually results in a smaller document size. The sample JSON_EMP table, with 42 documents contains 11,764 bytes of data, and when converted to BSON contains 11,362 bytes.

There will be occasions where the BSON format may be larger than the original JSON document. This expansion in size occurs when documents have many integer or numeric values. An integer type takes up 4 bytes in a BSON document. Converting a character value to a native integer format will take up more space but result in faster retrieval and comparison operations.

BSON_VALIDATE: Checking the Format of a Document

Db2 has no native JSON data type, so there no validation done against the contents of a BLOB column which contains JSON data. If the JSON object is under program control and you are using the JSON2BSON and BSON2JSON functions, you are probably not going to run across problems with the data.

If you believe that a document is corrupted for some reason, you can use the BSON_VALIDATE to make sure it is okay (or not!). The function will return a value of 1 if the record is okay, or a zero otherwise. The one row that we have inserted into the TESTJSON table should be okay.

```
SELECT BSON_VALIDATE(JSON_FIELD) FROM TESTJSON;
```

```
1
-----------
          1
```

The following SQL will inject a bad value into the beginning of the JSON field to test the results from the BSON_VALIDATE function.

```
UPDATE TESTJSON
  SET JSON_FIELD = BLOB('!') || JSON_FIELD;


SELECT BSON_VALIDATE(JSON_FIELD) FROM TESTJSON;

1
-----------
          0
```

Retrieving JSON Documents

The last section described how we can insert and retrieve entire JSON documents from a column in a table. This section will explore several functions that allow access to individual fields within the JSON document. These functions are:

- JSON_VAL – Extracts data from a JSON document into SQL data types
- JSON_TABLE – Returns a table of values for a document that has array types in it
- JSON_TYPE – Returns the data type of a specific field within a JSON document
- JSON_LEN – Returns the count of elements in an array type inside a document
- JSON_GET_POS_ARR_INDEX – Retrieve the index of a value within an array type in a document

Sample JSON Table Creation

The following SQL will load the JSON_EMP table with several JSON objects. These records are modelled around the SAMPLE database EMPLOYEE table.

```
CREATE TABLE JSON_EMP
  (
  SEQ INT NOT NULL GENERATED ALWAYS AS IDENTITY,
  EMP_DATA BLOB(4000) INLINE LENGTH 4000
  );
```

The following command will load the records into the JSON_EMP table. Only the first INSERT is displayed, but there is a total of 42 records included. A Db2 script file (DB2-V11-JSON-Examples.sql) containing all the JSON examples can be found in the same directory as this eBook. See the Appendix for more details on how to get a copy of this file.

```
INSERT INTO JSON_EMP(EMP_DATA) VALUES JSON2BSON(
    '{
    "empno":"000070",
    "firstnme":"EVA",
    "midinit":"D",
    "lastname":"PULASKI",
    "workdept":"D21",
    "phoneno":[7831,1422,4567],
    "hiredate":"09/30/2005",
    "job":"MANAGER",
    "edlevel":16,
    "sex":"F",
    "birthdate":"05/26/2003",
    "pay": {
      "salary":96170.00,
      "bonus":700.00,
      "comm":2893.00
      }
     }');
```

Additional JSON_DEPT Table

In addition to the JSON_EMP table, the following SQL will generate a table called JSON_DEPT that can be used to determine the name of the department an individual works in.

```
CREATE TABLE JSON_DEPT
  (
  SEQ INT NOT NULL GENERATED ALWAYS AS IDENTITY,
  DEPT_DATA BLOB(4000) INLINE LENGTH 4000
  );

INSERT INTO JSON_DEPT(DEPT_DATA) VALUES
  JSON2BSON('{"deptno":"A00", "mgrno":"000010", "admrdept":"A00",
              "deptname":"SPIFFY COMPUTER SERVICE DIV."}'),
  JSON2BSON('{"deptno":"B01", "mgrno":"000020", "admrdept":"A00",
              "deptname":"PLANNING"                    }'),
  JSON2BSON('{"deptno":"C01", "mgrno":"000030", "admrdept":"A00",
              "deptname":"INFORMATION CENTER"          }'),
  JSON2BSON('{"deptno":"D01",                   "admrdept":"A00",
              "deptname":"DEVELOPMENT CENTER"          }'),
  ...
  JSON2BSON('{"deptno":"J22",                   "admrdept":"E01",
              "deptname":"BRANCH OFFICE J2"            }');
```

JSON_VAL: Retrieving Data from a BSON Document

Now that we have inserted some JSON data into a table, this section will explore the use of the JSON_VAL function to retrieve individual fields from the documents. This built-in function will return a value from a document in a format that you specify. The ability to dynamically change the returned data type is extremely important when we examine index creation in another section.

The JSON_VAL function has the format:

```
JSON_VAL(document, field, type);
```

JSON_VAL takes 3 arguments:

- document – BSON document
- field – The field we are looking for (search path)
- type – The return type of data being returned

The search path and type must be constants – they cannot be variables so their use in user-defined functions is limited to constants.

A typical JSON record will contain a variety of data types and structures as illustrated by the following record from the JSON_EMP table.

```
{
  "empno":"200170",
  "firstnme":"KIYOSHI",
  "midinit":"",
  "lastname":"YAMAMOTO",
  "workdept":"D11",
  "phoneno":[2890],
  "hiredate":"09/15/2005",
  "job":"DESIGNER",
  "edlevel":16,
  "sex":"M",
  "birthdate":"01/05/1981",
  "pay": {
    "salary":64680.00,
    "bonus":500.00,
    "comm":1974.00
  }
}
```

There are number of fields with different formats, including strings (firstnme), integers (edlevel), decimal (salary), date (hiredate), a number array (phoneno), and a structure (pay). JSON data can consist of nested objects, arrays and very complex structures. The format of a JSON object is checked when using the JSON2BSON function and an error message will be issued if it does not conform to the JSON specification.

The JSON_VAL function needs to know how to return the data type back from the JSON record, so you need to specify what the format should be. The possible formats are:

Code	Format
n	DECFLOAT
i	INTEGER
l	BIGINT (notice this is a lowercase L)
f	DOUBLE
d	DATE
ts	TIMESTAMP (6)
t	TIME
s:n	A VARCHAR with a size of n being the maximum
b:n	A BINARY value with n being the maximum
u	An integer with a value of 0 or 1.

Retrieving Atomic Values

This first example will retrieve the name and salary of the employee whose employee number is "200170"

```
SELECT JSON_VAL(EMP_DATA,'lastname','s:20'),
       JSON_VAL(EMP_DATA,'pay.salary','f')
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

1                    2
-------------------- ------------------------------------------
YAMAMOTO                                                  64680
```

If the size of the field being returned is larger than the field specification, you will get a NULL value returned, not a truncated value.

```
SELECT JSON_VAL(EMP_DATA,'lastname','s:7')
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

1
--------------------
-
```

In the case of character fields, you may need to specify a larger return size and then truncate it to get a subset of the data.

```
SELECT LEFT(JSON_VAL(EMP_DATA,'lastname','s:20'),7)
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

1
-------
YAMAMOT
```

Retrieving Array Values

Selecting data from an array type will always give you the first value (element zero). The employees all have extension numbers but some of them have more than one. Some of the extensions start with a zero so since the column is being treated as an integer you will get only 3 digits. It's probably better to define it as a character string rather than a number!

```
SELECT JSON_VAL(EMP_DATA, 'phoneno', 'i') FROM JSON_EMP;

1
-----------
       3978
```

If you specify ":na" after the type specifier, you will get an error if the field is an array type. Hopefully you already know the format of your JSON data and can avoid having to check to see if arrays exist. What this statement will tell you is that one of the records you were attempting to retrieve was an array type. In fact, all the phone extensions are being treated as array types even though they have only one value in many cases.

```
SELECT JSON_VAL(EMP_DATA, 'phoneno', 'i:na') FROM JSON_EMP;

SQL20556N  The operation failed because multiple result values
cannot be returned from a scalar function "SYSIBM.JSON_VAL".
SQLSTATE=22547
```

If you need to access a specific array element in a field, you can use the "dot" notation after the field name. The first element starts at zero. If we select the 2nd element (.1) all the employees that have a second extension will have a value retrieved while the ones who don't will have a null value.

Retrieving Structured Fields

Structured fields are retrieved using the same dot notation as arrays. The field is specified by using the "field.subfield" format and these fields can be an arbitrary number of levels deep.

The pay field in the employee record is made up of three additional fields.

```
"pay": {
   "salary":64680.00,
   "bonus":500.00,
   "comm":1974.00
}
```

To retrieve these three fields, you need to explicitly name them since retrieving pay alone will not work.

```
SELECT JSON_VAL(EMP_DATA,'pay.salary','i'),
       JSON_VAL(EMP_DATA,'pay.bonus','i'),
       JSON_VAL(EMP_DATA,'pay.comm','i')
  FROM JSON_EMP
WHERE
  JSON_VAL(EMP_DATA,'empno','s:6') = '200170';

1           2           3
----------- ----------- -----------
      64680         500        1974
```

If you attempt to retrieve the pay field, you will end up with a NULL value, not an error code. The reason for this is that the JSON_VAL function cannot format the field into an atomic value so it returns the NULL value instead.

Detecting NULL Values in a Field

To determine whether a field exists, or has a null value, you need use the "u" flag. If you use the "u" flag, the value returned will be either:

- 1 – The field exists, and it has a value (not null or empty string)
- 0 – The field exists, but the value is null or empty
- null – The field does not exist

In the JSON_EMP table, there are a few employees who do not have middle names. The following query will return a value or 1, 0, or NULL depending on whether the middle name exists for a record.

```
SELECT JSON_VAL(EMP_DATA,'lastname','s:30'),
       JSON_VAL(EMP_DATA,'midinit','u')
FROM JSON_EMP;
```

The results contain 40 employees who have a middle initial, and two that do not. The results can be misleading because an employee can have the midinit field defined, but no value assigned to it:

```
{
 "empno":"000120",
 "firstnme":"SEAN",
 "midinit":"",
 "lastname":"O''CONNELL",...
}
```

In this case, the employee does not have a middle name, but the field is present. To determine whether an employee does not have a middle name, you will need to check for a NULL value (the field does not exist, or the field is empty) when retrieving the middle initial (9 rows):

```
SELECT COUNT(*) FROM JSON_EMP
  WHERE JSON_VAL(EMP_DATA,'midinit','s:40') IS NULL;
```

If you only want to know how many employees have the middle initial field (midinit) that is empty, you need to exclude the records that do not contain the field (7 rows):

```
SELECT COUNT(*) FROM JSON_EMP
  WHERE JSON_VAL(EMP_DATA,'midinit','s:40') IS NULL AND
        JSON_VAL(EMP_DATA,'midinit','u') IS NOT NULL;
```

Joining JSON Tables

You can join tables with JSON columns by using the JSON_VAL function to compare two values:

```
SELECT JSON_VAL(EMP_DATA,'empno','s:6') AS EMPNO,
       JSON_VAL(EMP_DATA,'lastname','s:20') AS LASTNAME,
       JSON_VAL(DEPT_DATA,'deptname','s:30') AS DEPTNAME
  FROM JSON_EMP, JSON_DEPT
WHERE
  JSON_VAL(DEPT_DATA,'deptno','s:3') =
  JSON_VAL(EMP_DATA,'workdept','s:3')
FETCH FIRST 5 ROWS ONLY;
```

```
EMPNO  LASTNAME             DEPTNAME
------ -------------------- ------------------------------
000010 HAAS                 SPIFFY COMPUTER SERVICE DIV.
000020 THOMPSON             PLANNING
000030 KWAN                 INFORMATION CENTER
```

```
000050 GEYER                 SUPPORT SERVICES
000060 STERN                 MANUFACTURING SYSTEMS
```

You need to ensure that the data types from both JSON functions are compatible for the join to work properly. In this case, the department number and the work department are both returned as 3-byte character strings. If you decided to use integers instead or a smaller string size, the join will not work as expected because the conversion will result in truncated or NULL values.

If you plan on doing joins between JSON objects, you may want to consider created indexes on the documents to speed up the join process. More information on the use of indexes is found at the end of this chapter.

JSON Data Types

If you are unsure of what data type a field contains, you can use the JSON_TYPE function to determine the type before retrieving the field.

The JSON_TYPE function has the format:

```
ID = JSON_TYPE(document, field, 2048);
```

JSON_TYPE takes 3 arguments:

- document – BSON document
- field – The field we are looking for (search path)
- search path size – 2048 is the required value

The 2048 specifies the maximum length of the field parameter and should be left at this value.

When querying the data types within a JSON document, the following values are returned.

ID	TYPE	ID	TYPE
1	Double	10	Null
2	String	11	Regular Expression
3	Object	12	Future use
4	Array	13	JavaScript
5	Binary data	14	Symbol
6	Undefined	15	Javascript (with scope)

ID	TYPE	ID	TYPE
7	Object id	16	32-bit integer
8	Boolean	17	Timestamp
9	Date	18	64-bit integer

The next SQL statement will create a table with standard types within it.

```
CREATE TABLE TYPES
  (DATA BLOB(4000) INLINE LENGTH 4000);

INSERT INTO TYPES VALUES
  JSON2BSON(
  '{
  "string"    : "string",
  "integer"   : 1,
  "number"    : 1.1,
  "date"      : {"$date": "2016-06-20T13:00:00"},
  "boolean"   : true,
  "array"     : [1,2,3],
  "object"    : {type: "main", phone: [1,2,3]}
  }');
```

The following SQL will generate a list of data types and field names found within this document.

```
SELECT 'STRING',JSON_TYPE(DATA, 'string', 2048) FROM TYPES
UNION ALL
SELECT 'INTEGER',JSON_TYPE(DATA, 'integer', 2048) FROM TYPES
UNION ALL
SELECT 'NUMBER',JSON_TYPE(DATA, 'number', 2048) FROM TYPES
UNION ALL
SELECT 'DATE',JSON_TYPE(DATA, 'date', 2048) FROM TYPES
UNION ALL
SELECT 'BOOLEAN', JSON_TYPE(DATA, 'boolean', 2048) FROM TYPES
UNION ALL
SELECT 'ARRAY', JSON_TYPE(DATA, 'array', 2048) FROM TYPES
UNION ALL
SELECT 'OBJECT', JSON_TYPE(DATA, 'object', 2048) FROM TYPES;

1         2
--------- -----------
ARRAY               4
BOOLEAN             8
DATE                9
NUMBER              1
INTEGER            16
STRING              2
OBJECT              3
```

Extracting Fields Using Different Data Types

The following sections will show how we can get atomic (non-array) types out of the JSON documents. We are not going to be specific which documents we want aside from the field we want to retrieve.

A temporary table called SANDBOX is used throughout these examples:

```
CREATE TABLE SANDBOX (DATA BLOB(4000) INLINE LENGTH 4000);
```

JSON INTEGERS and BIGINT

Integers within JSON documents are easily identified as numbers that don't have a decimal place in them. There are two different types of integers supported within Db2 and are identified by the size (number of digits) in the number itself.

- Integer – a set of digits that do not include a decimal place. The number cannot exceed −2,147,483,648 to 2,147,483,647
- Bigint – a set of digits that do not include a decimal place but exceed that of an integer. The number cannot exceed – 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

You don't explicitly state the type of integer that you are using. The system will detect the type based on its size.

The JSON_TYPE function will return a value of 16 for integers and 18 for a large integer (BIGINT). To retrieve a value from an integer field you need to use the "i" flag and "l" (lowercase L) for big integers.

This first SQL statement will create a regular integer field.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"count":9782333}');
```

The JSON_TYPE function will verify that this is an integer field (Type=16).

```
SELECT JSON_TYPE(DATA,'count',2048) AS TYPE
  FROM SANDBOX;

TYPE
-----------
         16
```

You can retrieve an integer value with either the 'i' flag or the 'l' flag. This first SQL statement retrieves the value as an integer.

```
SELECT JSON_VAL(DATA,'count','i') FROM SANDBOX;

1
-----------
    9782333
```

We can ask that the value be interpreted as a BIGINT by using the 'l' flag, so JSON_VAL will expand the size of the return value.

```
SELECT JSON_VAL(DATA,'count','l') FROM SANDBOX;

1
--------------------
             9782333
```

The next SQL statement will create a field with a BIGINT size. Note that we don't need to specify anything other than have a very big number!

```
DELETE FROM SANDBOX;

INSERT INTO SANDBOX VALUES
  JSON2BSON('{"count":94123512223422}');
```

The JSON_TYPE function will verify that this is a big integer field (Type=18).

```
SELECT JSON_TYPE(DATA,'count',2048) AS TYPE
  FROM SANDBOX;

TYPE
-----------
         18
```

We can check to see that the data is stored in the document as a BIGINT by using the JSON_TYPE function.

```
SELECT JSON_TYPE(DATA,'count',2048) FROM SANDBOX;

1
-----------
         18
```

Returning the data as an integer type 'i' will fail since the number is too big to fit into an integer format. Note that you do not get an error message - a NULL value gets returned.

```
SELECT JSON_VAL(DATA,'count','i') FROM SANDBOX;

1
-----------
          -
```

Specifying the 'l' flag will make the data be returned properly.

```
SELECT JSON_VAL(DATA,'count','l') FROM SANDBOX;

1
--------------------
      94123512223422
```

Since we have an integer in the JSON field, we also have the option of returning the value as a floating-point number (f) or as a decimal number (n). Either of these options will work with integer values.

```
SELECT JSON_VAL(DATA,'count','n') AS DECIMAL,
       JSON_VAL(DATA,'count','f') AS FLOAT
FROM SANDBOX;

DECIMAL                                             FLOAT
---------------------------------- ----------------------
                    94123512223422  9.41235122234220E+013
```

JSON NUMBERS and FLOATING POINT

JSON numbers are recognized by Db2 when there is a decimal point in the value. Floating point values are recognized using the Exx specifier after the number which represents the power of 10 that needs to be applied to the base value. For instance, 1.0E01 is the value 10.

The JSON type for numbers is 1, whether it is in floating point format or decimal format.

The SQL statement below inserts a salary into the table (using the standard decimal place notation).

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"salary":92342.20}');
```

The JSON_TYPE function will verify that this is a numeric field (Type=1).

```
SELECT JSON_TYPE(DATA,'salary',2048) AS TYPE
  FROM SANDBOX;

TYPE
-----------
          1
```

Numeric data can be retrieved in either number (n) formant, integer (i - note that you will get truncation), or floating point (f).

```
SELECT JSON_VAL(DATA,'salary','n') AS DECIMAL,
       JSON_VAL(DATA,'salary','i') AS INTEGER,
       JSON_VAL(DATA,'salary','f') AS FLOAT
FROM SANDBOX;

DECIMAL                        INTEGER     FLOAT
------------------------------ ----------- ----------------------
            92342.199999999997       92342  9.23422000000000E+004
```

You may wonder why number format (n) results in an answer that has a fractional component that isn't exactly 92342.20. The reason is that Db2 is converting the value to DECFLOAT(34) which supports a higher precision number, but can result in fractions that can't be accurately represented within the binary format. Casting the value to DEC(9,2) will properly format the number.

```
SELECT DEC(JSON_VAL(DATA,'salary','n'),9,2) AS DECIMAL
FROM SANDBOX;

DECIMAL
-----------
   92342.20
```

A floating-point number is recognized by the Exx specifier in the number. The BSON function will tag this value as a number even though you specified it in floating point format. The following SQL inserts the floating value into the table.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"salary":9.2523E01}');
```

The JSON_TYPE function will verify that this is a floating-point field (Type=1).

```
SELECT JSON_TYPE(DATA,'salary',2048) AS TYPE
  FROM SANDBOX;

TYPE
-----------
          1
```

The floating-point value can be retrieved as a number, integer, or floating-point value.

```
SELECT JSON_VAL(DATA,'salary','n') AS DECIMAL,
       JSON_VAL(DATA,'salary','i') AS INTEGER,
       JSON_VAL(DATA,'salary','f') AS FLOAT
FROM SANDBOX;

DECIMAL                          INTEGER     FLOAT
-------------------------- ----------- ----------------------
       92.522999999999996          92  9.25230000000000E+001
```

JSON BOOLEAN VALUES

JSON has a data type which can be true or false (Boolean). Db2 doesn't have an equivalent data type for Boolean, so we need to retrieve it as an integer or character string (true/false).

The JSON type for Boolean values is 8.

The SQL statement below inserts a true and false value into the table.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"valid":true, "invalid":false}');
```

We will double-check what type the field is in the JSON record.

```
SELECT JSON_TYPE(DATA,'valid',2048) AS TYPE
  FROM SANDBOX;

TYPE
-----------
          8
```

To retrieve the value, we can ask that it be formatted as an integer or number.

```
SELECT JSON_VAL(DATA,'valid','n') AS TRUE_DECIMAL,
       JSON_VAL(DATA,'valid','i') AS TRUE_INTEGER,
       JSON_VAL(DATA,'invalid','n') AS FALSE_DECIMAL,
       JSON_VAL(DATA,'invalid','i') AS FALSE_INTEGER
FROM SANDBOX;

TRUE_DECIMAL TRUE_INTEGER FALSE_DECIMAL FALSE_INTEGER
------------ ------------ ------------- -------------
           1            1             0             0
```

You can also retrieve a Boolean field as a character or binary field, but the results are not what you would expect with binary.

```
SELECT JSON_VAL(DATA,'valid','s:5') AS TRUE_STRING,
       JSON_VAL(DATA,'valid','b:2') AS TRUE_BINARY,
       JSON_VAL(DATA,'invalid','s:5') AS FALSE_STRING,
       JSON_VAL(DATA,'invalid','b:2') AS FALSE_BINARY
FROM SANDBOX;

TRUE_STRING TRUE_BINARY FALSE_STRING FALSE_BINARY
----------- ----------- ------------ ------------
true        0801        false        0800
```

JSON DATE, TIME, and TIMESTAMPS

This first SQL statement will insert a JSON field that uses the $date modifier.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"today":{"$date":"2016-07-01T12:00:00"}}');
```

Querying the data type of this field using JSON_VAL will return a value of 9 (date type).

```
SELECT JSON_TYPE(DATA,'today',2048) FROM SANDBOX;

1
-----------
          9
```

If you decide to use a character string to represent a date, you can use either the "s:x" specification to return the date as a string, or use "d" to have it displayed as a date. This first SQL statement returns the date as a string.

```
INSERT INTO SANDBOX VALUES
  JSON2BSON('{"today":"2016-07-01"}');

SELECT JSON_VAL(DATA,'today','s:10') FROM SANDBOX;

1
----------
2016-07-01
```

Using the 'd' specification will return the value as a date.

```
SELECT JSON_VAL(DATA,'today','d') FROM SANDBOX;

1
----------
2016-07-01
```

What about timestamps? If you decide to store a timestamp into a field, you can retrieve it in a variety of ways. This first set of SQL statements will retrieve it as a string.

```
INSERT INTO SANDBOX VALUES
   JSON2BSON('{"today":"' || VARCHAR(NOW()) || '"}');

SELECT JSON_VAL(DATA,'today','s:30') FROM SANDBOX;

1
------------------------------
2016-09-17-06.27.00.945000
```

Retrieving it as a Date will also work, but the time portion will be removed.

```
SELECT JSON_VAL(DATA,'today','d') FROM SANDBOX;

1
----------
2016-09-17
```

You can also ask for the timestamp value by using the 'ts' specification. Note that you can't get just the time portion unless you use a SQL function to cast it.

```
SELECT JSON_VAL(DATA,'today','ts') FROM SANDBOX;

1
--------------------------
2016-09-17 06:27:00.945000
```

To force the value to return just the time portion, either store the data as a time value (HH:MM:SS) string or store a timestamp and use the TIME function to extract just that portion of the timestamp.

```
SELECT TIME(JSON_VAL(DATA,'today','ts')) FROM SANDBOX;

1
--------
06:27:00
```

JSON Strings

For character strings, you must specify what the maximum length is. This example will return the size of the lastname field as 10 characters long.

```
SELECT JSON_VAL(DATA, 'lastname', 's:10') FROM JSON_EMP;

1
----------
HAAS
```

You must specify a length for the 's' parameter otherwise you will get an error from the function. If the size of the character string is too large to return, then the function will return a null value for that field.

```
SELECT JSON_VAL(DATA, 'lastname', 's:8') FROM JSON_EMP;

1
--------
HAAS
```

JSON_TABLE Function

The following query works because we do not treat the field phoneno as an array:

```
SELECT JSON_VAL(DATA, 'phoneno', 'i') FROM JSON_EMP;

1
-----------
       3978
```

By default, only the first number of an array is returned when you use JSON_VAL. However, there will be situations where you do want to return all the values in an array. This is where the JSON_TABLE function must be used.

The format of the JSON_TABLE function is:

```
   JSON_TABLE(document, field, type)
```

The arguments are:

- document – BSON document
- field – The field we are looking for
- type – The return type of data being returned

JSON_TABLE returns two columns: Type and Value. The type is one of a possible 18 values found in the table below. The Value is the actual contents of the field.

ID	TYPE	ID	TYPE
1	Double	10	Null
2	String	11	Regular Expression
3	Object	12	Future use
4	Array	13	JavaScript
5	Binary data	14	Symbol
6	Undefined	15	Javascript (with scope)
7	Object id	16	32-bit integer
8	Boolean	17	Timestamp
9	Date	18	64-bit integer

The TYPE field is probably something you wouldn't require as part of your queries since you are already specifying the return type in the function.

The format of the JSON_TABLE function is like JSON_VAL except that it returns a table of values. You must use this function as part of FROM clause and a table function specification. For example, to return the contents of the phone extension array for just one employee (000230) we can use the following JSON_TABLE function.

```
SELECT PHONES.* FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
WHERE JSON_VAL(E.EMP_DATA,'empno','s:6') = '000230';

TYPE        VALUE
----------- --------------------
         16 2094
         16 8999
         16 3756
```

The TABLE(...) specification in the FROM clause is used for table functions. The results that are returned from the TABLE function are treated the same as a traditional table.

To create a query that gives the name of every employee and their extensions would require the following query.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
PHONES.VALUE AS PHONE
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES;

LASTNAME   PHONE
---------- --------------------
HAAS       3978
THOMPSON   3476
THOMPSON   1422
```

```
KWAN       4738
GEYER      6789
STERN      6423
STERN      2433
PULASKI    7831
PULASKI    1422
PULASKI    4567
```

Only a subset of the results is shown above, but you will see that there are multiple lines for employees who have more than one extension.

The results of a TABLE function must be named (AS ...) if you need to refer to the results of the TABLE function in the SELECT list or in other parts of the SQL.

You can use other SQL operators to sort or organize the results. For instance, we can use the ORDER BY operator to find out which employees have the same extension. Note how the TABLE function is named PHONES and the VALUES column is renamed to PHONE.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
       PHONES.VALUE AS PHONE
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
ORDER BY PHONE;
```

```
LASTNAME   PHONE
---------- --------------------
THOMPSON   1422
PULASKI    1422
SCHNEIDER  1422
O'CONNELL  1533
MEHTA      1533
ALONZO     1533
SCOUTTEN   1682
ORLANDO    1690
```

You can even found out how many people are sharing extensions! The HAVING clause tells Db2 to only return groupings where there is more than one employee with the same extension.

```
SELECT PHONES.VALUE AS PHONE, COUNT(*) AS COUNT
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
GROUP BY PHONES.VALUE HAVING COUNT(*) > 1
ORDER BY PHONES.VALUE
```

```
PHONE                COUNT
-------------------- -----------
1422                           3
1533                           3
```

```
1793                           2
2103                           2
2167                           2
2890                           2
3332                           2
3780                           2
```

❷ In subsequent releases of Db2, the JSON_TABLE function will be replaced with the SQL standards version. The standards-based JSON_TABLE function is completely different than the current Db2 implementation, so you must make sure to use the correct function path (SYSTOOLS or SYSIBM) when you use this function in the future.

JSON_LEN Function

The previous example showed how we could retrieve the values from within an array of a document. Sometimes an application needs to determine how many values are in the array itself. The JSON_LEN function is used to figure out what the array count is.

The format of the JSON_LEN function is:

```
   count = JSON_LEN(document,field)
```

The arguments are:

- document – BSON document
- field – The field we are looking for
- count – Number of array entries or NULL if the field is not an array

If the field is not an array, this function will return a null value, otherwise it will give you the number of values in the array. In our previous example, we could determine the number of extensions per person by taking advantage of the JSON_LEN function.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
       JSON_LEN(E.EMP_DATA, 'phoneno') AS PHONE_COUNT
  FROM JSON_EMP E;

LASTNAME   PHONE_COUNT
---------- -----------
HAAS                 1
THOMPSON             2
KWAN                 1
GEYER                1
STERN                2
PULASKI              3
HENDERSON            1
```

```
SPENSER                 1
```

JSON_GET_POS_ARR_INDEX Function

The JSON_TABLE and JSON_LEN functions can be used to retrieve all the values from an array, but searching for a specific array value is difficult to do. One way to search array values is to extract everything using the JSON_TABLE function.

```
SELECT JSON_VAL(E.EMP_DATA, 'lastname', 's:10') AS LASTNAME,
       PHONES.VALUE AS PHONE
  FROM JSON_EMP E,
       TABLE( JSON_TABLE(E.EMP_DATA,'phoneno','i') ) AS PHONES
WHERE PHONES.VALUE = 1422;
```

An easier way to search an array is by using the JSON_GET_POS_ARR_INDEX function. This function will search array values without having to extract the array values with the JSON_TABLE function.

The format of the JSON_GET_POS_ARR_INDEX function is:

```
   element = JSON_GET_POS_ARR_INDEX(document, field)
```

The arguments are:

- document – BSON document
- field – The field we are looking for and its value
- element – The first occurrence of the value in the array

The format of the field argument is "{field:value}" and it needs to be in BSON format. This means you needs to add the JSON2BSON function around the field specification.

```
  JSON2BSON( '{"field":"value"}' )
```

This function only tests for equivalence and the data type should match what is already in the field. The return value is the position within the array that the value was found, where the first element starts at zero.

In our JSON_EMP table, each employee has one or more phone numbers. The following SQL will retrieve all employees who have the extension 1422:

```
SELECT JSON_VAL(EMP_DATA, 'lastname', 's:10') AS LASTNAME
  FROM JSON_EMP
```

```
WHERE JSON_GET_POS_ARR_INDEX(EMP_DATA,
  JSON2BSON('{"phoneno":1422}')) >= 0;

LASTNAME
----------
THOMPSON
PULASKI
SCHNEIDER
```

If we used quotes around the phone number, the function will not match any of the values in the table.

Updating JSON Documents

There are a couple of approaches available to updating JSON documents. One approach is to extract the document from the table in a text form using BSON2JSON and then using string functions or regular expressions to modify the data.

The other option is to use the JSON_UPDATE statement. The syntax of the JSON_UPDATE function is:

```
   JSON_UPDATE(document, '{$set: {field:value}}')
```

The arguments are:

- document – BSON document
- field – The field we are looking for
- value – The value we want to set the field to

There are three possible outcomes from using the JSON_UPDATE statement:

- If the field is found, the existing value is replaced with the new one
- If the field is not found, the field:value pair is added to the document
- ❷If you use $unset, and the value is set to the null keyword, the field is removed from the document

The field can specify a portion of a structure, or an element of an array using the dot notation. The following SQL will illustrate how values can be added and removed from a document.

A single record that contains 3 phone number extensions are added to a table:

```
INSERT INTO SANDBOX VALUES
   JSON2BSON('{"phone":[1111,2222,3333]}');
```

To add a new field to the record, the JSON_UPDATE function needs to specify the field and value pair.

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $set: {"lastname":"HAAS"}}');
```

Retrieving the document shows that the lastname field has now been added to the record.

```
SELECT BSON2JSON(DATA) FROM SANDBOX;

1
----------------------------------------------
{"phone":[1111,2222,3333],"lastname":"HAAS"}
```

If you specify a field that is an array type and do not specify an element, you will end up replacing the entire field with the value.

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $set: {"phone":9999}}');

SELECT BSON2JSON(DATA) FROM SANDBOX;

1
----------------------------------
{"phone":9999,"lastname":"HAAS"}
```

Running the SQL against the original phone data will work properly.

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $set: {"phone.0":9999}}');

SELECT BSON2JSON(DATA) FROM SANDBOX;

1
------------------------------------------------------------
{"phone":[9999,2222,3333]}
```

❷To delete a field, you must use $unset instead of $set and use null as the value for the field. To remove the lastname field from the record:

```
UPDATE SANDBOX
  SET DATA =
    JSON_UPDATE(DATA,'{ $unset: {"lastname":null}}');
```

Indexing JSON Documents

Db2 supports computed indexes, which allows for the use of functions like JSON_VAL to be used as part of the index definition. For instance, searching for an employee number will result in a scan against the table if no indexes are defined:

```
SELECT JSON_VAL(EMP_DATA, 'lastname', 's:20') AS LASTNAME
  FROM JSON_EMP
WHERE  JSON_VAL(EMP_DATA, 'empno', 's:6') = '000010';

LASTNAME
--------------------
HAAS


Cost = 13.628412

    Rows
  Operator
    (ID)
    Cost
   1.68
  RETURN
   (  1)
  13.6284
     |
   1.68
  TBSCAN
   (  2)
  13.6283
     |
    42
 Table:
 BAKLARZ
 JSON_EMP
```

To create an index on the empno field, we use the JSON_VAL function to extract the empno from the JSON field.

```
CREATE INDEX IX_JSON ON JSON_EMP
  (JSON_VAL(EMP_DATA,'empno','s:6'));
```

Rerunning the SQL results in the following explain plan:

```
Cost = 6.811412

          Rows
        Operator
          (ID)
          Cost
         1.68
        RETURN
         (  1)
        6.81141
          |
         1.68
        FETCH
         ( 2)
        6.8113
       /      \
    1.68        42
   IXSCAN    Table:
    ( 3)     BAKLARZ
 0.00484089  JSON_EMP
     |
    42
  Index:
  BAKLARZ
  IX_JSON
```

Db2 can now use the index to retrieve the record.

Simplifying JSON SQL Inserts and Retrieval

From a development perspective, you always need to convert documents to and from JSON using the BSON2JSON and JSON2BSON functions. There are ways to hide these functions from an application and simplify some of the programming.

One approach to simplifying the conversion of documents between formats is to use INSTEAD OF triggers. These triggers can intercept transactions before they are applied to the base tables. This approach requires that we create a view on top of an existing table.

The first step is to create the base table with two copies of the JSON column. One will contain the original JSON character string while the second will contain the converted BSON. For this example, the JSON column will be called INFO, and the BSON column will be called BSONINFO.

The use of two columns containing JSON would appear strange at first. The reason for the two columns is that Db2 expects the BLOB column to contain binary data. You cannot insert a character string (JSON) into the BSON column without converting it first. Db2 will raise an error so the JSON column is there to avoid an error while the conversion takes place.

From a debugging perspective, we can keep both the CLOB and BLOB values in this table if we want. The trigger will set the JSON column to null after the BSON column has been populated.

```
CREATE TABLE BASE_EMP_TXS (
  SEQNO    INT NOT NULL GENERATED ALWAYS AS IDENTITY,
  INFO     VARCHAR(4000),
  BSONINFO BLOB(4000) INLINE LENGTH 4000
  );
```

To use INSTEAD OF triggers, a view needs to be created on top of the base table. Note that we explicitly use the SYSTOOLS schema to make sure we are getting the correct function used here.

```
CREATE OR REPLACE VIEW EMP_TXS AS
  (SELECT SEQNO, BSON2JSON(BSONINFO) AS INFO FROM BASE_EMP_TXS);
```

At this point we can create three INSTEAD OF triggers to handle insert, updates and deletes on the view.

On INSERT the DEFAULT keyword is used to generate the ID number, the JSON field is set to NULL and the BSON column contains the converted value of the JSON string.

```
CREATE OR REPLACE TRIGGER I_EMP_TXS
  INSTEAD OF INSERT ON EMP_TXS
  REFERENCING NEW AS NEW_TXS
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  INSERT INTO BASE_EMP_TXS VALUES (
     DEFAULT,
     NULL,
     SYSTOOLS.JSON2BSON(NEW_TXS.INFO)
   );
END
```

On UPDATES, the sequence number remains the same, and the BSON field is updated with the contents of the JSON field.

```
CREATE OR REPLACE TRIGGER U_EMP_TXS
  INSTEAD OF UPDATE ON EMP_TXS
  REFERENCING NEW AS NEW_TXS OLD AS OLD_TXS
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE BASE_EMP_TXS
     SET (INFO, BSONINFO) = (NULL,
         SYSTOOLS.JSON2BSON(NEW_TXS.INFO))
     WHERE
       BASE_EMP_TXS.SEQNO = OLD_TXS.SEQNO;
END
```

Finally, the DELETE trigger will just remove the row.

```
CREATE OR REPLACE TRIGGER D_EMP_TXS
  INSTEAD OF DELETE ON EMP_TXS
  REFERENCING OLD AS OLD_TXS
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  DELETE FROM BASE_EMP_TXS
     WHERE
       BASE_EMP_TXS.SEQNO = OLD_TXS.SEQNO;
END
```

Applications will only deal with the EMP_TXS view. Any inserts will use the text version of the JSON and not have to worry about using the JSON2BSON function since the underlying INSTEAD OF trigger will take care of the conversion.

The following insert statement only includes the JSON string since the sequence number will be generated automatically as part of the insert.

```
INSERT INTO EMP_TXS(INFO) VALUES (
    '{
    "empno":"000010",
    "firstnme":"CHRISTINE",
    "midinit":"I",
    "lastname":"HAAS",
    "workdept":"A00",
    "phoneno":[3978],
    "hiredate":"01/01/1995",
    "job":"PRES",
    "edlevel":18,
    "sex":"F",
    "birthdate":"08/24/1963",
    "pay" : {
      "salary":152750.00,
      "bonus":1000.00,
      "comm":4220.00}
    }');
```

Selecting from the EMP_TXS view will return the JSON in a readable format:

```
SELECT SEQNO, CAST(LEFT(INFO,50) AS VARCHAR(50)) FROM EMP_TXS;

SEQNO       2
----------- --------------------------------------------------
          1 {"empno":"000010","firstnme":"CHRISTINE","midinit"}
```

The base table only contains the BSON but the view translates the value back into a readable format.

An update statement that replaces the entire string works as expected.

```
UPDATE EMP_TXS SET INFO = '{"empno":"000010"}' WHERE SEQNO = 1;
SELECT SEQNO, CAST(LEFT(INFO,50) AS VARCHAR(50)) FROM EMP_TXS;

SEQNO       2
----------- --------------------------------------------------
          1 {"empno":"000010"}
```

If you want to manipulate the BSON directly (say change the employee number), you need to refer to the BASE table instead.

```
UPDATE BASE_EMP_TXS
  SET BSONINFO = JSON_UPDATE(BSONINFO,
    '{$set: {"empno":"111111"}}')
  WHERE SEQNO = 1;
```

And we can check it using our original view.

```
SELECT SEQNO, CAST(LEFT(INFO,50) AS VARCHAR(50)) FROM EMP_TXS;

SEQNO       2
----------- --------------------------------------------------
          1 {"empno":"111111"}
```

Summary

The current Db2 11.1 release (and Db2 10.5) includes several user-defined functions (UDFs) that were originally designed to be used by internal JSON interfaces. Based on feedback from several customers, we've documented what is used internally within Db2. While these routines have not been officially published, they are available for customer use.

Note: Remember that these functions may change in the future to conform to the SQL standard.

A

Appendix

Additional Resources for Db2

Resources to Build Your Db2 Skills

Rely on the wide range of IBM experts, programs, and services that are available to help you take your Information Management skills to the next level. Participate in our online community through developerWorks. Find tutorials, articles, whitepapers, videos, demos, best practices, Db2 Express downloads, and more. Visit ibm.com/developerworks/data.

IBM Certification Exams

Find industry-leading professional certification exams, including new certifications for Db2 10.5 with BLU Acceleration and Db2 11.1:

- Db2 10.5 Database Administration Upgrade Exam (Exam 311)
- Db2 11.1 DBA for LUW (Exam 600)
- Db2 10.5 Fundamentals for LUW (Exam 615)

Visit ibm.com/certify for more information and exam availability.

IBM Training

IBM is committed to helping our clients achieve the skills and expertise to take their careers to the next level. We offer a comprehensive portfolio of technical training and education services designed for individuals, companies, and public organizations to acquire, maintain, and optimize their IT skills in IBM Software and IBM Systems. Visit ibm.com/software/data/education for details and course availability.

BigData University

Learn about Db2 and various big data technologies at your pace and at your place. Big Data University offers helpful online courses with instructional videos and exercises to help you master new concepts. Course completion is marked with a final exam and a certificate. Visit bigdatauniversity.com.

Information Management Bookstore

Find the most informative Db2 books on the market, along with valuable links and offers to save you money and enhance your skills. Visit http://bit.ly/DB2_Books.

IBM Support for Db2

Access the IBM Support Portal to find technical support information for Db2 11.1, including downloads, notifications, technical documents, flashes, and more. Visit ibm.com/support.

Want a glimpse into the future? Check out the new support experience beta - https://ibm.biz/support-pilot.

Look for answers to your Db2 questions? Please try dWAnswers forum - http://ibm.biz/dwAnswersDB2.

IBM Data Magazine

The magazine's mission is to deliver substantive, high-quality content on the latest data management developments and IBM advances, as well as create a strong community of the world's top information management professionals.

IBM Data magazine vividly demonstrates how the smart use of data and information advances broad business success, providing the context that enables data management professionals at all levels to make more informed choices and create innovative, synchronized, agile solutions. See more at: ibmdatamag.com.

International Db2 User Group (IDUG)

IDUG is all about community. Through education events, technical resources, unique access to fellow users, product developers and solution providers, they offer an expansive, dynamic technical support community. IDUG delivers quality education, timely information and peer-driven product training and utilization that enable Db2 users to achieve organizational business objectives, and to drive personal career advancement. Visit idug.org.

Join the Conversation

Stay current as Db2 11.1 evolves by using social media sites to connect to experts and to contribute your voice to the conversation. Visit one or more of the following:

- https://twitter.com/IBM_DB2
- https://facebook.com/DB2community
- http://bit.ly/BLUVideos
- http://linkd.in/DB2Professional
- https://twitter.com/ibmbigdata
- http://www.planetdb2.com/
- http://developer.ibm.com/data/db2/

Additional eBook Material

The DB2DEMO program lets you try out many of the features that are found in this eBook. The program and the corresponding installation manual are found in the same directory that this eBook is found in:

https://ibm.ent.box.com/v/DB2v11eBook

To use DB2DEMO, you must be running on a Windows workstation and be able to connect to a Db2 server. If you want to try Db2 on your workstation, you can download the free Db2 Developer Community edition from:

http://www.ibm.com/us-en/marketplace/ibm-db2-direct-and-developer-editions

Examples that use Jupyter Notebooks can be found on our Github repository at: github.com/DB2-Samples

Any updates to the eBook and demonstration software will be found in this directory. If you have any comments or suggestions, please contact the authors at:

George Baklarz: baklarz@ca.ibm.com
Enzo Cialini: ecialini@ca.ibm.com

Db2 for Linux, Unix, and Windows

Version 11 Highlights

George Baklarz and Enzo Cialini

The Db2 11.1 release delivers several significant enhancements including Database Partitioning Feature (DPF) for BLU columnar technology, improved pureScale performance and High Availability Disaster Recovery (HADR) support, and numerous SQL features.

This book was written to highlight many of the new features and functions that are now available in this release, without you having to search through various forums, blogs, and online manuals. We hope that this book gives you more insight into what you can now accomplish with Db2 11.1, and include it on your shortlist of databases to deploy, whether it is on premise, in the cloud, or in virtualized environments.

Coverage Includes:

- An overview of the new packaging changes in Db2 11.1, along with pre-requisites for installing the product
- Enhancements to Db2 BLU, including support for BLU columnar tables in a DPF (Data Partitioning Feature) environment
- pureScale capabilities that provide continuous availability in a production environment, along with all the enhancements that have been made to simplify installation and management of a cluster
- SQL and compatibility enhancements

George Baklarz, B. Math, M. Sc., Ph.D. Eng., has spent 31 years at IBM working on various aspects of database technology. George has written 10 books on Db2 and other database technologies. George is currently part of the Worldwide Core Database Technical Sales Team.

Enzo Cialini, B.Sc., is a Senior Technical Staff Member and Master Inventor in the Worldwide Core Database Technical Sales Team and formerly the Chief Quality Assurance Architect for Db2 & PureData in the IBM Toronto Db2 Development Team. He is also a published book author and written various papers on Db2. Enzo has 25 years of experience in database technology, software development, testing, support, competitive analysis and production deployments.