TIBCO ActiveSpaces®

Concepts

Version 4.8.0
June 2022

Contents

About This Product

The TIBCO ActiveSpaces® software is a distributed in-memory data grid product. Some features of ActiveSpaces® include use of familiar database concepts, high I/O capacity, and network scalability.

ActiveSpaces features a complete redesign and reimplementation of the product and is straightforward to understand, use, and administer.

Product Editions

ActiveSpaces is now available in two editions: Community Edition and Enterprise Edition.

	Community Edition	Enterprise Edition
Ideal for	Getting started with ActiveSpaces for implementing application projects, including proof of concept projects, for testing, and for deploying applications in a production environment. Production deployments running up to 5 nodes (a total of the copyset nodes or proxies in your data grid) For more information, see Terms used in Community and Enterprise Editions.	All application development projects, and for deploying and managing applications in the production environment of an enterprise. Production deployments with more than 5 nodes (a total of the copyset nodes or proxies in your data grid) For more information, see Terms used in Community and Enterprise Editions.
Features	All features of the Enterprise Edition except enterprise monitoring using dashboards.	Includes all features presented in this documentation set.
Limitations	Run up to 5 nodes (a total of the copyset nodes or proxies in your data grid). Although the community license limits the number of production instances, you can	No limitations on a total of the copyset nodes or proxies in your data grid.

	easily upgrade to the enterprise edition as your use of ActiveSpaces expands.	
Cost	Free	Paid
Compatibility	Compatible with both the enterprise and community editions of TIBCO FTL®	Depends on the enterprise edition of TIBCO FTL for monitoring and management of data grid components and secure communication.
TIBCO Support	No access to TIBCO Support	Access to TIBCO Support

Terms used in Community and Enterprise Editions

- Node - a copyset node or proxy where each copyset node or proxy is an operating system process with a unique process ID.
- Process ID - For the purposes of the definition of Node, Process ID means a standard computer industry term that uniquely identifies each operating system process.
- Copyset - For the purposes of the definition of Node, “Copyset” means a logical grouping of nodes such that a portion of the data is shared uniformly by all the nodes that form a copyset.

Overview of TIBCO ActiveSpaces

ActiveSpaces software is a distributed in-memory data grid product.

To lift the burden of big data, ActiveSpaces provides a distributed in-memory data grid that can increase processing speed to reduce reliance on costly transactional systems. ActiveSpaces provides an infrastructure for building highly scalable, fault-tolerant applications. It creates virtual data caches from the aggregate memory of participating nodes, scaling automatically as nodes join the data grid. By combining the features and performance of databases, caching systems, and messaging software, it supports very large, highly volatile data sets and event-driven applications.

Why ActiveSpaces?

A traditional RDBMS fails to keep up with the growing volume of data and the high rate of I/O operations per second. This drawback of RDBMS can impact the performance and slow down the system.

ActiveSpaces is ideal for enterprises that handle a large amount of data or have a high volume of I/O activities per second. ActiveSpaces provides horizontal scalability, where you have the flexibility to segregate the data across a group of computers. For example, if you have 25000 operations per second, they can be divided across 10 computers that handle 2500 operations per second. Or, if your enterprise needs to store 10 TB of data, it can be distributed across 5 computers that contribute 2 TB each to store the data.

What Is ActiveSpaces?

ActiveSpaces uses the concepts of grid computing to provide a scalable, distributed, and durable data grid. The data grid serves as a system of record to store terabytes of data in an enterprise. ActiveSpaces provides a fast, consistent, and fault-tolerant system that supports a high rate of I/O operations in a scalable manner.

For more information about the ActiveSpaces concepts, see the Grid Computing in ActiveSpaces section.

Benefits of ActiveSpaces

ActiveSpaces offers many advantages as compared with a traditional RDBMS.

The following list highlights the benefits of using ActiveSpaces:

- ActiveSpaces performance and customer experience
- Requires minimum investment because the system scales on low cost commodity or virtualized hardware
- Updates data and systems continually and provides immediate and accurate response
- Supports many hardware and software platforms, so programs running on different kinds of computers in a network can communicate seamlessly
- Scales linearly and transparently when nodes are added (An increase in the number of nodes produces a corresponding increase in the memory and processing power available to the data grid)
- Enables smooth and continuous working of your application without code modification or restarts
- Provides location transparency without the hassles of determining where or how to store data and search for it
- Notifies applications automatically as soon as data is modified

Attributes of ActiveSpaces

These attributes of ActiveSpaces set it apart from traditional RDBMS.

Scalability

The biggest advantage of using ActiveSpaces is scalability. You can scale up the system horizontally to hold terabytes of data without bringing the system down. You also have complete administrative control over data redistribution.

System of Record

ActiveSpaces serves as a distributed, large-scale system of record that spans across nodes in an enterprise. The system of record uses the concepts of the traditional RDBMS such as table, rows, and columns. In addition, every node saves a portion of the data locally in a fault-tolerant and durable manner.

Faster Access to Data

ActiveSpaces support queries and indexes that improve performance. Queries run faster because data is cached in-memory. Queries in ActiveSpaces are a subset of the SQL language. The filtering and indexing capabilities offered by ActiveSpaces expedite the execution of queries.

TIBCO FTL® for Secure Communication

ActiveSpaces uses the capabilities of TIBCO FTL® 6.1.0 or later. A specific FTL realm contains configuration information and connectivity parameters for communication between the ActiveSpaces data grid processes and client applications. ActiveSpaces uses TIBCO FTL for the following key tasks:

- Communication between application programs and the data grid.
- Internal communication among data grid component processes
- Configuration, monitoring, and management of data grid components

Note: With TIBCO FTL 6.1.0 or later, ActiveSpaces uses the realm service capabilities or processes of the TIBCO FTL server. In this documentation, the term "realm service" is used to refer to TIBCO FTL 5.x realm server or TIBCO FTL 6.x realm service.

For the versions of TIBCO FTL that are compatible with ActiveSpaces, see the *readme.txt* file.

High-Performance ACID-Compliant Data Grid

The data grid provides atomicity, consistency, isolation, and durability (ACID) of data. ACID-compliance is achieved by using transactions and concurrency control across multiple tables.

Transaction Isolation

A transaction comprises a set of operations that can modify the content of the data grid. Owing to transaction isolation, an ongoing transaction does not affect the queries that are being run on the content. For example, if another system is trying to read a row that you are trying to modify, that system either gets the data before the modification or it gets the data after the modification. The system does not get partially committed transactions. Even if the transaction is distributed over a network that involves multiple rows and tables, transaction isolation ensures that there are no uncommitted reads

(dirty reads).

To achieve the highest level of transaction isolation, pessimistic transactions are used. This guarantees that a row is consistently accessed by the operation that initially accessed the row until the transaction commits or rolls back. This blocks any operation that can violate database consistency or isolation. Transactions take care of rolling back partially committed transactions.

Easy-to-Use APIs

ActiveSpaces provides tools for data definitions that are akin to the SQL language. You can also define how data is distributed across a configurable number of nodes. The support functions in the API are easy to use. You can use the functions to retrieve metadata information about the data grid, a specific table, or a result set.

Real-Time Push Events

ActiveSpaces provides real-time push events over the network to servers and client applications to change the data grid. Table listeners receive data change events through callback notifications.

Cloud Ready

It is easy to deploy ActiveSpaces on cloud, on-premises, or hybrid environments. You can easily build ActiveSpaces into microservices with container deployment products such as Docker.

Redesigned from the Ground Up

Since the 3.0 version of ActiveSpaces, ActiveSpaces software is completely redesigned and reimplemented to make it more user-friendly for both end users and administrators. ActiveSpaces 3.x is not backward compatible with the earlier versions of the product. ActiveSpaces 3.x is faster because it relies on TIBCO FTL for the underlying communication.

ActiveSpaces 3.x and later use the terminology of a traditional RDBMS. See the Comparison Matrix.

Terminology Used to Address the TIBCO FTL Realm

With TIBCO FTL 6.1 or later, ActiveSpaces uses the realm service capabilities or processes of the TIBCO FTL server. The following changes are made to the terminology to generically address the components of TIBCO FTL 5.x and TIBCO FTL 6.x:

The Term Used in the Document	The Equivalent Component in TIBCO FTL 5.4.1	The Equivalent Component in TIBCO FTL 6.1 or Later
Realm service	Realm server	Realm service running on the TIBCO FTL server
Realm service URL	Realm server URL	TIBCO FTL server URL
Backup realm service	Backup realm server	TIBCO FTL server that is a member of a cluster of three or more TIBCO FTL servers
Primary Realm	Primary Realm Server and its Backup Realm Server	A cluster of primary TIBCO FTL servers that provide realm services for the data grid.
Satellite Realm	Satellite Realm Server and its Backup Realm Server	A cluster of satellite TIBCO FTL servers that are connected to a cluster of primary TIBCO FTL servers.

Grid Computing in ActiveSpaces

ActiveSpaces uses grid computing to bring together computers in your network that can contribute their processing power, memory, and storage to solve a complex problem. ActiveSpaces uses grid computing concepts to store and process the contents of a data grid.

What Is a Data Grid?

ActiveSpaces stores data in data grids. In a data grid, data is stored in the form of tables. A data grid is equivalent to a database of a traditional RDBMS.

Tables

A table comprises multiple rows that are spread out in the data grid. The tables are similar to the tables in a traditional RDBMS, made of rows and columns. Unlike the traditional RDBMS where all the data in the table reside on one computer, a data grid segregates the table row-wise and stores the rows in different ActiveSpaces processes called nodes.

Rows

Like the traditional RDBMS, a row comprises a set of columns and is uniquely identified by the primary index. A row becomes the unit of measurement for the data grid. Rows are distributed across the data grid. When scaling up, the data grid controls where a newly added row must be stored.

Columns

A row is made of a collection of columns. Every column uniquely identifies a piece of information. Every column has a type and a value associated with it. For example, the Employee Name column is of data type String and has the value "Joe Smith".

A column can be of the following data types:

- `long`
- `double`
- `string`

- `datetime`
- `opaque`

Primary Index

Uniquely identifies a row in a table. It is equivalent to a primary key in a traditional RDBMS. You can have more than one column that forms a primary index.

Secondary Index

Is similar to a primary index but can refer to multiple rows in a table. A secondary index comprises one or more columns of a table and is used to efficiently retrieve the rows of a table by reducing the number of rows scanned for retrieval by queries. Without a secondary index, this would involve a full table scan to identify which rows match the query. With a secondary index, additional space is used to help speed up the query and quickly identify matching rows without a full-table scan.

Primary and secondary indexes can be of the data types: long, string, and datetime.

How Is the Data Stored in a Data Grid?

Unlike traditional RDBMS, a data grid is not stored in one place. An ActiveSpaces data grid leverages the storage capacity and computing power from multiple computers.

To understand how the data is stored, you must first familiarize yourself with the following concepts:

Nodes

A node is an ActiveSpaces process running within a computer. The node holds a portion of the data forming the data grid both in memory and on disk. The smallest unit of data held by a node is a row. Other than storing data of a row, the node is also responsible for handling requests to read or update the row. As a result, the data spanning across a group of nodes collectively form a data grid.

Nodes can be run from a physical computer, a virtual machine, or a Docker container.

Persistence on Nodes ActiveSpaces supports the Shared Nothing mode of persistence where every node saves its data locally to the disk.

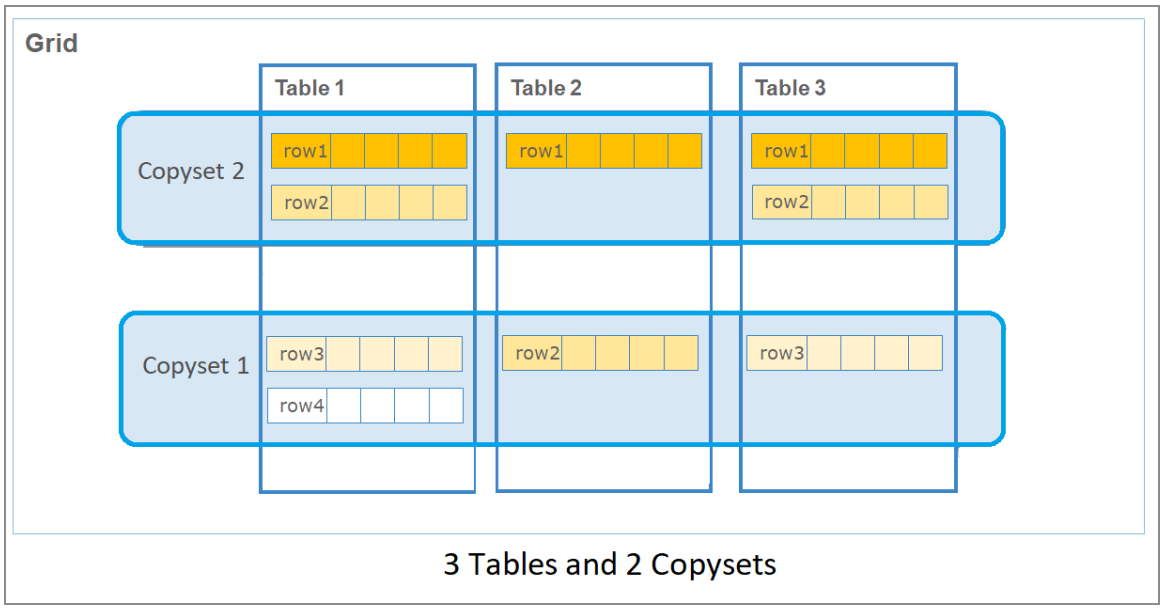
Copysets

Copysets are logical grouping of nodes such that a portion of the data is shared uniformly by all the nodes that form a copyset. This ensures fault tolerance. Every node in the copyset, also known as the replica, has an identical copy of the data. For example, assume that a row (R1) comprises employee name, employee ID, and department. There are nodes, N1, N2, and N3 in copyset1. N1, N2, and N3 store identical copies of R1. When you add new data or request for an update on a row in a copyset, the update is written to all the nodes in the copyset before acknowledging the success of the operation. Keeping the nodes of a copyset on different computers helps prevent data loss during system failures.

Copysets help you scale your data horizontally. When you add a new copyset to a data grid, you can redistribute the existing data to the new nodes of that copyset, thereby distributing the load on the data grid with the help of the newly added copyset.

The following image is a logical diagram showing how rows of three tables are distributed across two copysets in the data grid.

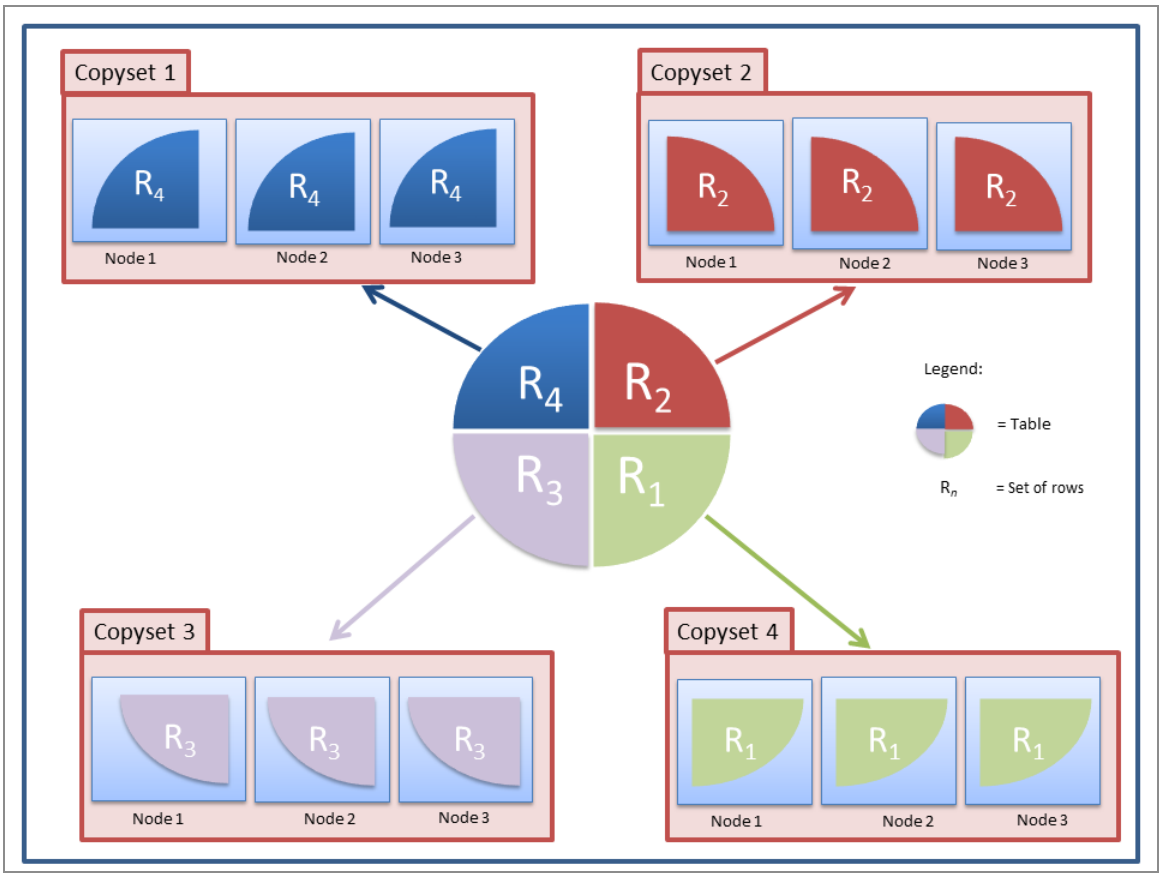
Rows Distributed Across Copysets



3 Tables and 2 Copysets

In the following image, the rows of a table are broken down into four sets (each owned by a different copyset). The nodes running in a given copyset are identical replicas of each other.

How One Table is Distributed in a Data Grid with Four Copysets and Three Nodes



To understand more about sizing a copyset and a data grid, see Sizing Guide.

Primary Node

When a copyset has more than one node in a copyset, one of the nodes is the primary node, which stores data and provides read access. The other nodes in the copyset are secondary nodes that store backup copies of the data. The key role of the primary node is to interact with the proxy process. The primary node receives the client operation and replicates it to the other nodes in the copyset. The client operation is applied in parallel at the primary node and all secondary nodes. The primary node is responsible for sharing the result of the request with the proxy.

If the primary node goes down for some reason, one of the other nodes in the copyset takes over as the primary node. Updates from client applications continue as usual without any loss of data because all of the data has been replicated from the original primary node to all of the other nodes in the copyset. The nodes of a copyset must

reside on different machines to ensure that one machine failure does not cause data loss.

Reasons for Using Multiple Nodes

There are several reasons for using multiple nodes:

- Nodes in different copysets are created with the goal of scaling horizontally. Thus, multiple copysets are created, each with a slice of the data.
- Nodes in the same copyset are created to provide multiple replicas for fault tolerance. These contain identical copies of the data.
- In a production environment, you might decide to use multiple nodes for a combination of reasons. For example, you might choose to have two replicas per copyset and multiple copysets (say three) to scale horizontally. In this example, your environment would have a total of six nodes.

To sum it up, the data is stored in copysets as described in the previous sections. The copysets put together form a data grid.

Replication

To replicate data, you must configure the copysets in the data grid such that `copyset_size` is greater than 1.

The `copyset_size` configuration setting applies to all copysets in the data grid. When the `copyset_size` is greater than 1, one node in each copyset acts the primary node that stores data and provides read access to that data. The other nodes in each copyset are secondary nodes that store copies of the data on the primary node. Every time data is written to the primary node, data is synchronized at the primary node and all secondary nodes in the copyset.

When the primary node of a copyset is down, one of the secondary nodes in the copyset takes over as the primary node. As each secondary node of the copyset contains copies of the same data that resides on the primary node, no data loss occurs and data grid operations continues as long as at least one node of the copyset remains running.

Processes in ActiveSpaces

The following processes are involved in creating, maintaining, and querying the data grid:

- ActiveSpaces Client Applications
- Proxy
- Realm Service
- State Keeper
- Node

TIBCO ActiveSpaces Client Applications

The client applications use the API libraries shipped with the product to build custom applications. Client applications interact with the data grid by using the proxy process.

Proxy

A proxy is a mediator between a client request and the data grid. Based on the client request, the proxy identifies the primary node in a copyset and interacts with the primary node till the request is processed and shared with the client. You can have many proxies in a data grid.

Realm Service

A data grid is run inside a TIBCO FTL realm. A TIBCO FTL realm serves as a repository for data grid configuration information and provides communication services that enable all data grid processes to communicate with each other.

A client application accesses the data grid by using the realm service URL. In TIBCO FTL 6.0.0 or later, the realm service URL is the URL of the TIBCO FTL server. The realm service offers the following capabilities:

- Stores data grid definitions
- Communicates with the administrative tools to store and retrieve data grid definitions
- Communicates with all the processes running in the data grid and updates the internal configuration if anything changes
- Collects monitoring data from all processes

Fault Tolerance in Realm Services Used in TIBCO 6.0.0 or Later

TIBCO FTL 6.1.0 or later uses a quorum-based fault tolerance mechanism. A cluster of at least three TIBCO FTL core servers must be run. Each core server provides a realm service. Those realm services all cooperate to provide fault tolerance for the data grid. Fault tolerance is assured as long as a quorum of servers is always running. Each core server must be run on a separate machine. Clients receive a list of URLs at which they can connect to those TIBCO FTL core servers.

State Keeper

A state keeper runs internally in the data grid and tracks all the data in the data grid. Each state keeper saves the data locally on the disk. When you start the realm service, the state keeper receives the data grid configuration information from the realm service. State keepers are responsible for the following functions:

- Tracking and managing all the copysets in a data grid
- Tracking the proxies in a data grid
- Identifying a primary node in each copyset
- Promoting one of the secondary nodes as primary, in case the primary node of a copyset goes down
- Ensuring consistency as the data grid scales up

Fault Tolerance in State Keepers

It is good practice to have three state keepers running in a production environment. A set of fault-tolerant state keeper processes protects the data grid's run time state information and ensures nonstop access to it. One of the state keepers is designated the lead state keeper and supplies this information to the proxies and copyset nodes. If the lead state keeper goes down, one of the secondary state keepers takes over as the lead. In a fault-tolerant set of three state keepers, a quorum of two state keepers must always be running to ensure data consistency in split brain scenarios. If a state keeper is restarted while a quorum is running, one of the running state keepers updates the state of the restarted state keeper. If the number of running state keepers falls below the quorum and the state of a copyset changes (for example, a node goes down), operations on the data grid fails. When this happens, the remaining state keepers must be brought down and then all state keepers must be restarted.

Node

For more information on nodes, see the "Nodes" section in How Is the Data Stored in a Data Grid?.

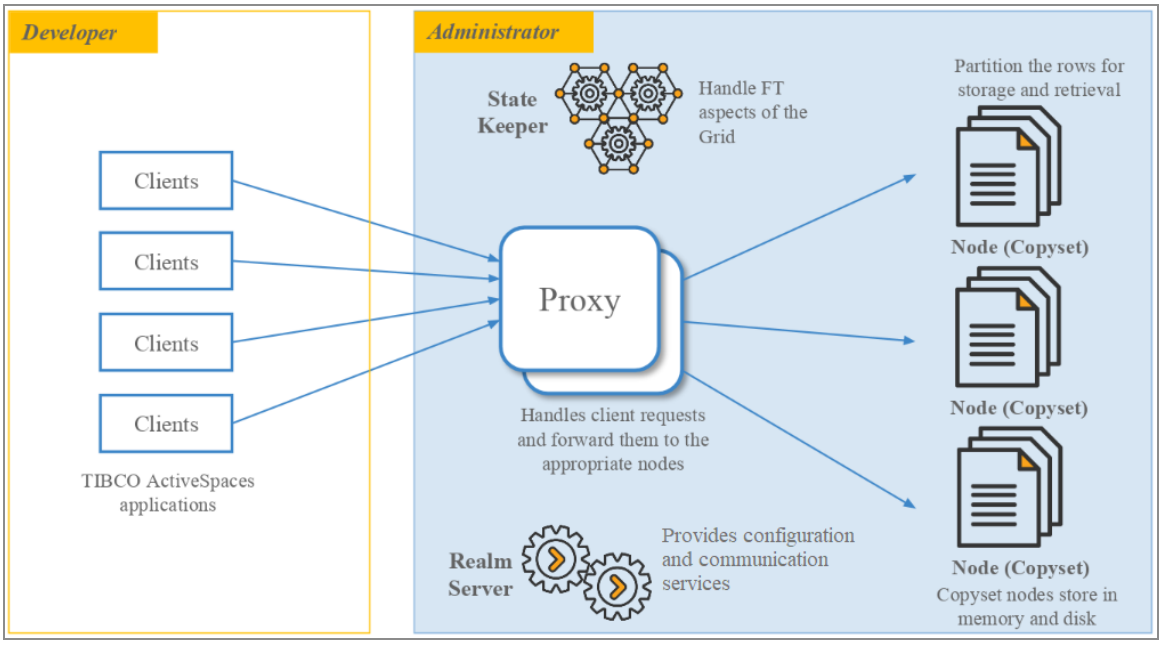
Fault Tolerance in Nodes

To prevent data loss, you can run up to three nodes per copyset. For production deployments, TIBCO recommends using at least two nodes per copyset.

The Workflow for a PUT Operation

A client application initiates a PUT request. The request reaches the proxy. Like all the ActiveSpaces processes, the proxy identifies the data grid by the data grid name and the realm service URL (In TIBCO FTL 6.0.0 or later, the realm service URL is the URL of the TIBCO FTL server). The proxy forwards the request to the appropriate primary node. The primary node handles the processing of the data. After all the secondary nodes are updated with the changes, the result is returned to the proxy and the proxy then shares the result with the client application. The realm service and the state keeper run outside of the operation datapath.

The Workflow



Log Levels

The log level determines the level of detail and the quantity of log statements. Typically, log levels must not be adjusted because producing excess log output can affect performance. However, there are situations such as debugging an issue where different log levels must be configured.

ActiveSpaces uses the logging mechanism provided by TIBCO FTL. For more information, see "Log Levels" in *TIBCO FTL® Development*.

The `tibdg` client library as well as the `tibdgkeeper`, `tibdgproxy`, and `tibdgnode` process can all be configured with nondefault log levels. The client library has an API used to set the log level. The data grid processes can be configured by using the `-t` command-line parameter. The log levels are set by using one of the following forms:

```
element:level
```

or

```
element:level;element:level;element:level
```

Often, additional debug log statements can be gathered by using `tibdg:debug` as the log level. The output of this command shows more log statements than the default log level (`tibdg:info`). The specific syntax when used with the `tibdgproxy` data grid process would be:

```
tibdgproxy -r http://realm_url:port -t tibdg:debug -n p_01
```

Other log levels or elements might be requested to be set when investigating specific issues as needed.

You can use the logs to trace client API calls on a thread basis. To trace the calls, use the client API to set the log level to `tibdgapi:debug3`. This triggers the client library to produce log statements for calls to API functions.

Transaction Isolation

ActiveSpaces enforces the highest level of transaction isolation: *serializable*. As a result, serialization can delay database operations as transactions wait for other transactions to commit or roll back.

ActiveSpaces uses a pessimistic transaction model: blocking any operations that can violate database consistency or isolation. For example, when an operation in transaction A refers to table row R, and an operation in a second transaction B also refers to row R, then the second operation blocks until transaction A either commits or rolls back. Similarly, an operation within a transaction can block operations in non-transacted sessions.

Checkpoints

ActiveSpaces checkpoints provide the ability to save the state and data of a running data grid. A checkpoint can then be used to restore a complete data grid on the same computer, to move the entire data grid, or to replicate a data grid to another data grid for disaster recovery.

The data collected by a checkpoint is guaranteed to be logically consistent across the entire data grid. A checkpoint does not contain the data from a partially committed transaction.

On creation, ActiveSpaces checkpoint performs the following activities:

- The realm database is saved.
- The configuration of the data grid in the realm is saved.
- Each state keeper's internal governing state information is saved.
- The relevant files needed to restore each node of a data grid are saved in the checkpoints subdirectory of each node's data directory.

Creating a checkpoint fails in the following scenarios:

- A realm is not reachable.
- A quorum of state keepers is not running.

Checkpoint Types

A manual checkpoint is created manually by using the `tibdg` administrative tool and a periodic checkpoint is created automatically by configuring the data grid to create periodic checkpoints.

Manual Checkpoints

- Initiated by using the `tibdg` administrative tool.
- Are given a unique name to help with checkpoint identification

Periodic Checkpoints

- Configured at the data grid level.
- Taken at a fixed interval while the data grid is running.
- Cannot be given a name.

Both manual and periodic checkpoints can be manually removed, and are subject to removal based on the retention setting.

Disaster Recovery

Disaster Recovery is a situation where a set of running systems must be replaced by another set of running systems due to failure, damage, loss of connectivity, or other traumatic event. Disaster Recovery is a large scale event and is not intended to replace fault tolerance where the failure of individual components can be recovered or otherwise accommodated without stopping a running system.

In a disaster recovery scenario, running systems are not expected to seamlessly or automatically failover to backup or alternative systems. Recovering from a disaster scenario implies a substantial and potentially large scale system stoppage and a restart of an entirely new instance of the previously running system. It is not intended for short term outages such as normal maintenance operations.

Note: The replacement systems activated during disaster recovery are designed to remain in operation for days, weeks, or even months depending on the severity of the disaster.

ActiveSpaces supports disaster recovery by creating a gridset.

Gridsets

The purpose of gridsets is to help set up the disaster recovery process. A group of data grids that share the same set of consistent data is referred to as a gridset. Each gridset has a name, which exists in the same namespace as data grid names (For example, you cannot have a data grid named “prod” and a gridset named “prod”). Each gridset also has a single primary data grid. Within a gridset, there is a single authoritative schema, which is owned by the primary data grid of the gridset.

Each data grid might belong to at most one gridset. Data grids in a gridset do not need to have the same replication factor, number of copysets, or number of state keepers, but care must be taken to ensure that a mirror data grid has sufficient capacity to handle the required load if administrators choose to make it the primary data grid.

Types of Data Grids

The following list differentiates the types of data grids in ActiveSpaces:

Stand-Alone Data Grids

Any data grid that does not belong to a gridset is a stand-alone data grid. All operations included in the ActiveSpaces API are permitted on stand-alone data grids.

Primary Data Grids

Any data grid that is listed as the primary of a gridset is a primary grid. All operations included in the ActiveSpaces API are permitted on primary grids. Primary grids are responsible for supplying mirror grids with data on request.

Mirror Data Grids

Any data grid that is included in a gridset, but is not currently the primary of that gridset is a mirror grid. Only read operations are allowed on mirror grids (For example, GET, queries, iterators, and so on). Read operations are executed against the most recent checkpoint that has been mirrored from the primary grid. Mirror grids are responsible for requesting updates from the primary grid.

Mirroring

The process by which data is copied from one data grid to another is called mirroring.

Data mirrored between data grids is a logical copy of the user data available on the primary grid and is copied to the mirror grid only if a checkpoint has been taken at the primary grid (either a user-created checkpoint or a periodic checkpoint causes mirroring). Until all copysets in the primary grid have confirmed that their data has been mirrored, data in the checkpoint being mirrored is not available.

Bulk Mirroring

If a mirror grid has no previous checkpoints available or if the primary grid has insufficient information to identify the rows that changed since the last mirrored checkpoint, bulk mirroring is used. During bulk mirroring, all rows present in the checkpoint being mirrored are sent to the mirror grid.

Incremental Mirroring

ActiveSpaces attempts to minimize the data sent between grids whenever possible by using incremental mirroring. When a mirror grid has a previous checkpoint as the

starting point, and the primary grid has sufficient information to identify all rows that changed, only rows that were updated or deleted are sent to the mirror grid.

Best Practices for a Development Environment

In many enterprises, programmers act as administrators during the development and test phases of a project.

To develop and test application programs that use ActiveSpaces software, deploy the following processes:

Processes	Numbers
Realm service	One
State keeper	One
Node	One
Proxy	One
Your application programs	Your application programs appropriate

In a development environment, you can run all of these processes on the same host computer.

Sample Scripts

Refer to the *TIBCO_HOME*/as/<version>/samples/readme.md before using the sample scripts.

The following scripts are available:

TIBCO_HOME/as/<version>/samples/scripts/as-start defines a simple data grid and starts its component processes.

TIBCO_HOME/as/<version>/samples/scripts/as-stop stops those component processes.

Sample Docker Environment

The docker-compose sample environment is provided to demonstrate how to deploy an ActiveSpaces data grid in Docker. For more information, see *TIBCO_HOME*/as/<version>/samples/docker/README.md.

Note: The installation environment of ActiveSpaces is referenced as *TIBCO_HOME*. For example, on Microsoft Windows, *TIBCO_HOME* might be `C:\tibco`.

When you are ready use ActiveSpaces to scale your data beyond one computer, you can create additional copysets and nodes in the data grid and run the nodes on separate computers.

Pre-Production Checklist

While developing or testing a new ActiveSpaces application, TIBCO recommends to evaluate each item in the following checklist to confirm expected behavior prior to moving the application to production. Some of these items may include simply understanding how to perform the specific activity while others may involve evaluating how specific failure scenarios may then lead to timeouts or other errors being returned to the application.

- **Rolling Upgrade** - At some point, an upgrade to a newer version of FTL and ActiveSpaces is needed. Either the grid can be stopped and all grid processes upgraded at the same time or the grid can be upgraded one process at a time in a rolling fashion as described in the steps in the documentation.
- **Monitoring** - The messaging monitoring stack (InfluxDB/Grafana) provides dashboards and stats collection for the different grid processes. This should be deployed and connected to the FTL server used by the grid. The FTL server pushes grid stats it collects through the tibmongateway to InfluxDB.
- **Single Node Failure** - A test should be done to stop the primary node in a copyset while live ops (gets/puts/deletes) are happening. The expected behavior would be that another node (an alive secondary node) takes over for the primary node that stopped. Client ops may be delayed or may experience a timeout exception during this transition period, which must be handled in the application as appropriate.
- **Proxy Failure** - A test should be done to stop a proxy while live ops (gets/puts/deletes) are happening. The expected behavior would be that the client application re-binds to a different proxy that should also be running. Client ops may be delayed or may experience a timeout exception during this transition period, which must be handled in the application as appropriate.
- **Client Application Error Handling** - Timeouts and other types of errors can occur in a distributed system and the client application should have error handling in place to address these scenarios (either by retrying, returning the error, etc).
- **Secondary Node Sync/Catchup** - At times a node may need to be stopped for an extended amount of time (due to host maintenance, etc) and that node is considered a dead secondary in the copyset as the primary node continues processing live operations. When the secondary node is restarted, it needs to complete a sync/catchup process where it is sent any data that was missed while it was stopped.

This can involve more read or write activity (disk, network, CPU) than is typical when just processing live operations (it also happens in parallel with ongoing live ops) so this scenario should be exercised on representative hardware prior to production.

- **Redistribution To New Copysets** - As a grid grows in size, it is often necessary to create a new copyset, which will then be given some of the existing data in the grid. An administrative command is used to begin the redistribution of data to the new copyset. This can involve more read and write activity (disk, network, CPU) than is typical when just processing live operations (it also happens in parallel with ongoing live ops). You must handle this scenario on representative hardware prior to production.
- **Live Backup/Restore or DR/Mirroring** - Exercise features like live backup and restore or DR/mirroring on representative hardware prior to production.
- **Log/Status Collection** - Collecting tibdg status, tibdg proxy status <proxy_name>, tibdg node status <node_name>, all log files for grid process and the FTL server, and the LOG file from the node data directory is often required to diagnose unexpected behavior. Automating this collection and exercising it prior to production is recommended.

Best Practices for a Production Environment

To use ActiveSpaces software in a production environment, deploy the following processes.

Processes	Minimum Required	Description
Realm service	Three (You need at least three processes to run a full quorum.)	TIBCO recommends that you run a fault-tolerant set of realm services. Run a quorum of realm services. Each realm service must be run on separate host computers.
State keeper	Three state keeper processes	To ensure high availability during a network partition or hardware failure, each state keeper process must run on a separate host computer. Not doing so might result in grid-wide data loss. At any given time, you must maintain a quorum of running state keepers. To run more than one state keeper, configure three state keepers and ensure you have at least two running state keepers.
Node	Two nodes per copyset	For greater data protection you can run three nodes per copyset. In a fault tolerant setup, if there are more than one node, one node acts as a primary node and the other nodes are secondary nodes. **Note: Note:** Additional copies can become expensive in two ways: Increasing the node count by one adds one complete copy of all the data. Every node process must run on a separate host computer. Usually this requirement determines the number of host computers you must maintain. For example, a data grid with three copysets and two nodes per copyset requires six nodes, all on separate hosts. Increasing to three nodes per copyset would require nine nodes, all on separate hosts.

Processes	Minimum Required	Description
Proxy	One proxy process	You can run additional proxies to increase the capacity for client programs and to improve response time. For best results, run proxy processes on a separate host computer.
Your application programs	Run as many processes as appropriate.	

Components Sharing a Host Computer

You can reduce number of host computers in a production environment by running more than one component per host.

For example, you can run a realm service, a state keeper, a node, and a proxy, all on one host. (In contrast, do not run two state keepers on the same host.) For effective fault tolerance, run the nodes of each copyset on separate host computers.

Warning: Combining component processes on a host computer increases the risk that a single point of failure on the host can disrupt all those processes simultaneously. Assess the risk tolerance of your enterprise.

Best Practices for Cloud Environments

For cloud environments, TIBCO recommends using a persistent, local Solid-State Drive (SSD) type that provides consistent performance and does not artificially throttle the Input Output Operations per Second (IOPS). An example of throttling IOPS is the burst throttling done by gp2 SSD types on AWS. For more information on different EBS volume types provided by Amazon, look for "Amazon EBS Volume Types" on https://docs.aws.amazon.com. For information on monitoring the performance of your EBS volume, look for "EBS Performance - I/O Characteristics and Monitoring" on https://docs.aws.amazon.com.

Programming with ActiveSpaces

These concepts and definitions pave the way to a more detailed understanding of applications programming with ActiveSpaces software.

Data Grid

A distributed database, including all the component processes that implement it.

Connection

An application program connects to a data grid. A `Connection` object is analogous to a traditional database connection.

Session

An application program interacts with a data grid through one or more `Session` objects. Each session insulates the data grid interactions within one program thread from the interactions in other threads.

A session can be transacted or non-transacted. GET, PUT, and DELETE operations in a transacted session occur within a transaction, and do not take effect until the program explicitly commits the transaction.

A session can be used to define the tables and indexes of the data grid by using SQL Data Definition Language (DDL) statements such as CREATE TABLE, DROP TABLE, CREATE INDEX, and DROP INDEX.

Table

An ActiveSpaces data grid organizes and presents data as rows in tables, like a traditional relational database.

Administrators define tables within the data grid.

Programs can GET a row from a table, PUT a row into a table, and DELETE a row from a table.

Programs can query a table for the rows that match a filter.

Primary Key

Each table distinguishes a primary key, or more briefly, the key.

Values of the key are unique: no two rows in a table have the same key value.

Secondary Index

A table can have zero or more secondary indexes, which facilitate queries. The `tibdg` tool can be used to create a secondary index on a table by using the `index create` command.

Iterator

An iterator is associated with a single table. An application can use an iterator to perform queries on a table. An iterator always returns entire rows from a table for its results.

Statement

A `Statement` is used to execute SQL statements. A `Statement` is not tied to a particular table. The table to act on is obtained from the SQL string used to create the `Statement` object. Statements can be used to execute SQL Data Manipulation Language (DML) statements. SQL DML statements include SELECT, INSERT, and so on.

ResultSet

A `ResultSet` contains the results of a SQL SELECT statement executed by using a `Statement` object. A `ResultSet` is used to iterate over the rows that satisfy the conditions of the SELECT statement. The columns of the rows in `ResultSet` are determined by the select list specified in the SQL SELECT statement.

Metadata

Metadata contains descriptive information about the data grid. There are two types of metadata: `GridMetadata` and `ResultSetMetadata`.

`GridMetadata` is retrieved from a `Connection` object. `GridMetadata` can be used to programmatically retrieve ActiveSpaces version information, the data grid name, and information about the tables that have been defined in the data grid. The table information includes the names of the tables that have been defined and also the information about the columns and indexes defined for each table.

`ResultSetMetadata` is retrieved from a `ResultSet` object. `ResultSetMetadata` can be used to find information about the columns in each row of a `ResultSet`. This column information includes the number of columns in a row, the labels and data types of the columns, whether or not a column can contain or return NULL values, and so on.

Structuring Programs

These steps outline the main structural components of most application programs that access an ActiveSpaces data grid. The steps assume that a table has already been configured for the data grid. When updating and querying data in ActiveSpaces, you can use `Table` objects or `Statement` objects or both in your application. `Table` objects provide a key/value interface to the data grid and `Statement` objects provide a SQL interface to the data grid.

An Overview of the Tasks

The following procedure summarizes the tasks ActiveSpaces application programs perform.

Procedure

1. Initialize the ActiveSpaces objects as listed in Task A: Initializing ActiveSpaces Objects.
2. On a specific table in the data grid, perform the appropriate operations as listed in Task B: Performing Data Grid Operations .
3. After you are done, clean up the objects as listed in Task C: Cleaning up and Closing the Connection.

Task A: Initializing ActiveSpaces Objects

Procedure

1. Initialize the ActiveSpaces library, if required.
   - C API - call `tibdg_Open()`
   - Java API - not required
   - Go API - not required
2. Connect to a data grid. For details, see Connection.
3. Create `Session` objects.

   See Session.
4. Create objects that stay open for the duration of the Connection.
   a. Open `Table` objects to execute key/value ops on the table. See Table.

b. Create `TableListener` objects to monitor events corresponding to changes in the table. See Table Listener.

c. Create `Statement` objects to support executing the same SQL more than once (commonly known as a `prepared` statement). See Statement.

Task B: Performing Data Grid Operations

Procedure

1. Access the data grid by using the appropriate APIs.
   a. Use key/value or iterator methods of a table object. See the "Table Operations" section in Table.
   b. Query the data grid by executing queries from SQL SELECT statements. See Statement.
   c. Modify the data grid by executing updates from SQL DML statements. See Statement.
2. Close any object created when accessing the data grid.
   a. Destroy all Row objects.
   b. Close all Iterator objects.
   c. Close all ResultSet objects.

Task C: Cleaning up and Closing the Connection

Procedure

1. Close all `Statement` objects.
2. Close all `TableListener` objects.
3. Close all `Table` objects.
4. Close all `Session` objects.
5. Close the data grid `Connection` object.

Connection

Programs begin their interactions with an ActiveSpaces data grid by first creating a `Connection` object. The `Connection` object can then be used to retrieve grid metadata or to create `Session` objects.

From the `Connection` object, a program can create one or many `Session` objects.

Grid Metadata

Grid metadata is retrieved from a `Connection` object by calling the get grid metadata API. Each time the `GridMetadata` is retrieved, the information returned reflects the current table information in the data grid.

A program must destroy the `GridMetadata` object after it has finished using it and before making any subsequent calls to retrieve updated `GridMetadata`.

To learn more about grid metadata, see the section on "Metadata" in Programming with ActiveSpaces.

Table Metadata

Table information is retrieved from the `GridMetadata` object as a `TableMetadata` object. A `TableMetadata` object is retrieved by using the table's name.

If the application program does not know the names of the tables that have been defined in the data grid, the `GridMetadata` object provides a method to get an array of all table names that have been defined. This array can then be used to get a single `TableMetadata` object.

A column name or index name is used to get information about a particular column or index from a `TableMetadata` object. If the application program does not have the names of the columns or indexes of a table, the `TableMetadata` object provides methods to get an array of the column names or index names.

A separate method to get the name of the primary index is provided by the `TableMetadata` object. The name of the primary index is then used to retrieve information about the columns of the primary index. The columns of the primary index make up the primary key of the table.

The `TableMetadata` object and strings retrieved from it do not have to be destroyed as these objects are owned by the `GridMetadata` object and are destroyed when the grid metadata is destroyed.

Session

Programs use sessions to insulate data grid operations within program threads and to group operations into atomic transactions.

Sessions and Threads

It is good practice to create a separate session for each program thread that accesses the data grid.

Programs must use sessions in a thread-safe way. That is, two threads must not simultaneously access the same session. Violating this constraint can yield unpredictable results.

Sessions and Transactions

Each session can be either transacted or non-transacted. Programs determine this semantic property when creating each session.

In a transacted session, all GET, PUT, UPDATE and DELETE operations occur within a transaction, which is bound to the session. The session implicitly starts the transaction. Programs explicitly call the session's commit and rollback methods. (As these methods complete, they automatically start a new transaction in the session.)

If a program operates within several open transactions simultaneously, use a separate session and thread for each transaction.

In a nontransacted session, GET, PUT, UPDATE, and DELETE operations are immediate: that is, when the method completes, the effect of the operation is also complete.

However, operations in a transacted session can block operations in a non-transacted session. For further explanation, see Transaction Isolation.

Only GET, PUT, UPDATE and DELETE operations are affected by a transacted session, the corresponding commit, and rollback APIs. Other commands such as iterators, queries, and DDL updates do not have different behavior when running on a transacted session versus a non-transacted session.

Sessions and Defining Tables

After a session has been created, it can be used to define tables programmatically. For more information, see "Defining a Table by Using SQL DDL Commands" in *TIBCO ActiveSpaces® Administration*.

Table

Table objects represent data grid tables within an application program.

Tables and Sessions

A program opens a table object by calling a session's open table method. The program can use the table object's methods to operate on the corresponding table in the data grid.

Opening a table object does not lock the table in the data grid.

If the session is transacted, then table operations occur within the session's transaction. Within a transaction you can interact with multiple tables.

If the session is non-transacted, then table operations are not transacted.

Table Operations

Tables support the following data grid operations:

- PUT a row into the table
- GET a row of the table
- UPDATE a row in the table.
- DELETE a row from the table
- Create an iterator to present the results of a table query

Primary Key

Every table requires a primary key, which can consist of one or more columns. The data type of primary key columns can be long, string, or datetime.

Examples of primary keys include employee number, invoice number, or MAC address.

The value of the primary key always remains unique across all the rows of the table. That is, database operations can never create two rows with the same key value; instead, they overwrite data in the existing row with that key value.

Creating Tables

Before a program can use a table or its rows for operations, the table must first be defined. A table can be defined programmatically by using a `Session` object or an administrator can define a table by using the ActiveSpaces administration tool. For details, see "Defining a

Table" and "Defining a Table by Using SQL DDL Commands" in *TIBCO ActiveSpaces® Administration*.

PUT

The PUT operation adds a row to a data grid table.

Before calling the `put()` method, your program must first create a row object and set its columns with values.

The row object must contain a value in all columns of the primary key. The value of the key is unique. If the table already contains a row with that key value, then the PUT operation replaces the existing row within the table. The PUT operation overwrites any unchanged columns in the row. The columns that are not part of the primary key can either contain data or be NULL.

GET

The GET operation retrieves a row of a data grid table.

Before calling the `get()` method, your program must first create a row object and set a value in all columns of the primary key. The value of the key is unique.

If the table contains a row with that key value, then the GET operation returns the contents of that row in a new row object. If the table does not contain a row with that key value, then the method returns null.

UPDATE

The UPDATE operation modifies rows that already exist in a data grid table. Before calling the `update()` method, you must first create a row object and set the primary key columns to uniquely identify the row that is going to be updated. Next, set any non-primary key columns that must be updated. Columns that are not set are not modified in the existing row.

NOTE: You cannot use the UPDATE operation to modify the primary key fields.

If a row with the primary key exists in the table, the `update()` method returns 1 whether or not any columns are updated. If a row with the primary key does not exist in the table, no update is done and 0 is returned.

DELETE

The DELETE operation deletes a row from a data grid table.

Before calling the `delete()` method, your program must first create a row object and set a value in all columns of the primary key. The value of the key is unique.

If the table contains a row with that key value, then the DELETE operation deletes that row from the table.

If the table does not contain a row with that key value, then the method returns without changing the table.

Iterator

A table iterator is used to iterate over all of the rows or a specific subset of the rows in the table. The create iterator operation submits a query on a data grid table and creates an iterator object to present the query results.

Supply a filter string as an argument to the create iterator operation. The filter string follows the syntax of the WHERE clause of a SQL SELECT statement excluding the WHERE keyword. All rows in the table for which the filter string evaluates to `true` are returned by the iterator.

An iterator object receives batches of matching rows from the data grid. The prefetch property of the iterator determines the batch size.

Properties can be set when an iterator is created thereby affecting the query behavior. For more information, see Data Consistency for Queries.

The iterator object presents the program with the individual rows that match the query one at a time.

To release resources within the data grid component processes, the program must close the iterator object and close each row object retrieved by using the iterator.

An implicit timeout limits the lifespan of iterator objects. Program calls that access an iterator after that timeout elapses return an error.

Avoid queries that result in full table scans, which can be resource-intensive and time-consuming.

Table Listener

A table listener is used to monitor events corresponding to changes in a table. A table listener is created from a specific table.

When you use a table listener, you can either monitor the contents of a specified table or a subset of rows in a specific table. For example, by using a filter, your application can track a table containing customer data and more specifically, can track the activity in a particular region to know when new customers are added or deleted, or when customers move to another region.

Events

An event indicates that the data in a table has changed.

An event can be of the following types:

- PUT: Indicates that new data has been added to the table. PUT is also used to indicate that existing data in the table has been updated.

  PUT events have a current value, which is a copy of the row that was added to or updated in the table. If the PUT operation replaces an existing row, or the UPDATE operation modifies an existing row, the event additionally has a previous value. The previous value is a copy of the row before the PUT or UPDATE operation.

- DELETE: Indicates that a row has been deleted from the table. DELETE events have a previous value, which is a copy of the row before the DELETE operation.

- ERROR: Indicates that something has happened in the system that indicates that the flow of events is disrupted. ERROR events have an error code and an error description. The application must destroy the table listener. Depending on the error code, it might or might not make sense for the application to re-create the table listener. The ActiveSpaces API documentation provides more details on the specific error codes that are possible.

- EXPIRED: Indicates that a row has expired. When rows are removed from a table due to expiration, table listeners on the table receive EXPIRED events when the expired rows match the table listeners' filters.

When creating a table listener, you must specify the table that is the source of the events of interest and a callback function that is invoked when events are delivered to the application. The callback function executes in a thread that is internal to the ActiveSpaces client library and is expected to complete in a timely fashion. The client library retains ownership of the events and the rows they contain so any data that is required outside of the callback must be copied and managed by the application itself.

Filtering Events

When the table listener is being created you can optionally specify a filter string to further narrow the scope of events received.

The filter string specifies the criteria that events must match in order for them to be delivered to the table listener. The filter is equivalent to the WHERE clause of a SQL SELECT statement, excluding the WHERE keyword, and is applied to both the current and previous values for the row that has changed.

For example, in a table containing customer data with a column called state, the filter `state = “CA”` limits the events delivered to only those involving customers in California.

Listening to Specific Event Types

When the table listener is being created, you can optionally provide a `Properties` object containing a list of event types that restrict the listener to only receiving events of those types. This feature is commonly used to listen to expired events, but to ignore any PUT or DELETE events that occur on the table.

The property name is `TIBDG_LISTENER_PROPERTY_STRING_EVENT_TYPE_LIST` and the property value must be a comma separated list of string event types. The valid choices are any combination of `"put"`, `"delete"`, and `"expired"`. For example, to listen to only expired events, you would use a property value of `"expired"`. To listen to both expired events and delete events, you would use a property value of `"delete,expired"`.

Statement

`Statement` objects are used to run SQL commands on the data grid. Queries (SELECT statements) and data manipulation language (DML) SQL commands can be run by using `Statement` objects. `Statement` objects are created by invoking the `createStatement()` method on the `Session` object.

A `Statement` object is created for each individual SQL command. A `Statement` can be run multiple times and must be closed when it is no longer needed.

You can create a query by using the SELECT statement, update rows by using an INSERT or an INSERT OR REPLACE statement, or delete rows by using a DELETE statement.

The INSERT statement is supported for both, transacted and non-transacted sessions. For more information about INSERT statement, see The SQL INSERT Statement.

For more information about INSERT OR REPLACE statement, see The INSERT OR REPLACE Statement.

The DELETE statement is supported for non-transacted sessions only. For more information about the DELETE statement, see The SQL DELETE Statement.

Properties

`Statement` properties affect the behavior of the statement. `Statement` properties can be set when a statement is first created or when a `Statement` is executed.

Examples of the properties that can be set are:

- Query prefetch - Number of rows to return in a batch when a query is first executed and each time more rows are requested while iterating through the results.
- Query fetch timeout - Number of seconds to wait for a batch of rows to be returned before the method waiting for the rows time out.

For specific information about `Statement` properties, see each language API's documentation for the following tasks:

- Creating a Statement from a `Session` object
- Executing a query by using a `Statement` object
- Executing a DML command by using a `Statement` object

Parameters

Parameters serve as placeholders for values in a SQL command. Parameters are used to separate the data of a SQL command from the command itself. This can be useful when the same command can be run multiple times by just varying the data of the command thereby increasing performance of the data grid. Parameters can be used to prevent SQL injection attacks in queries.

Parameters in a SQL command are specified by using `'?'` (question mark). For SELECT and DELETE statements, parameters are supported for the values of comparisons in `WHERE` clauses. For INSERT statements, parameters are supported for column values.

The `Statement` interface provides methods for setting the values of any parameters used in a SQL command. Separate methods for setting parameter values are provided for each data type supported by ActiveSpaces. The `setNull()` method is provided to specify that a parameter's value must be empty (SQL NULL). All parameter values must be specified

before running the statement or an error is returned. Parameters are numbered starting with 1.

Executing Statements

The `Statement` interface provides two methods for executing the statement. The `executeQuery()` method is used to execute statements, which have been created by using a SELECT command. The `executeQuery()` method returns a `ResultSet` object that contains the resulting rows of a query.

The `executeUpdate()` method is used to execute statements that have been created by using a DML command. The `executeUpdate()` method returns the number of rows that were successfully processed. If the wrong method is used to execute a statement, an error is returned. For information about the current DML commands supported, see Modifying Data in a Table.

ResultSet

A `ResultSet` contains the set of rows that make up the result of a query. A `ResultSet` object is returned when `executeQuery()` is invoked on a `Statement` created for a SELECT statement. The `ResultSet` object must be closed when it is no longer needed. A `ResultSet` object is returned even if no rows were found for the query.

The `ResultSet` object contains methods that allow you to iterate over the rows of the query result. A `ResultSet` object can be iterated over only once. The `hasNext()` method is used to check if there is a row that can be retrieved. The `next()` method is used to retrieve the next row object of the result.

Row Objects

A `Row` object retrieved from `ResultSet` contains the columns specified in the select list of a query. Each `Row` object retrieved from `ResultSet` must be closed when it is no longer needed. A `Row` object contains the methods to find out the data type of each of its columns, whether a column has a value, and the methods to retrieve a column's value by its data type.

The label of a column in the select list is used to access the data for each column. The columns of a row are accessed by using the label specified for the column in the select list using ActiveSpaces. For example,

```
SELECT col1 AS myname FROM table1
```

where `myname` is used as the column label.

If a label was not specified for a column in the select list, the column's name from the table is used as the label. For example,

```
SELECT col1 FROM table1
```

where `col1` is used as the column label.

If a label was not specified and the column of the select list is an expression, the expression string is used as the label. For example,

```
SELECT col1, date('now') FROM table1 WHERE col1 <= 10
```

where the expression string, `date('now')`, is used as the column label.

When a label is specified for a column, or the column is an expression, the label or expression must be used exactly as it was specified in the original query string and is case-sensitive. If a label was not specified for a column and the column is from the table, the label is not case-sensitive.

ResultSet MetaData

`ResultSetMetadata` can be used to discover information about the columns that comprise the rows of a query result. The `ResultSetMetadata` for a SELECT statement can be retrieved by invoking the `getResultSetMetadata()` method of the Statement object.

The `ResultSetMetadata` information includes the number of columns in a result row, each column's data type, the label given to the column, the name of the column from the table, and the name of the table for each column.

If a label was not specified for a column, the name of the column from the table is used as the label. If a label was not specified for a column and the column is not from a table, but is an expression that is calculated as part of the result, the entire expression string is used as the label. It is always safest to access a column of a `ResultSet` by its label as a `ResultSet` column always has a label but might not necessarily have a name from a table.

SQL Identifiers

TIBCO recommends administrators to define table and column names that follow the SQL identifier rules. For details, see "Column Names" in *TIBCO ActiveSpaces® Administration*.

In some situations, a data grid might contain non-standard table or column names. For example, a table copied from a legacy database might have columns with names that contain a space character or there might be a table or column name with a SQL keyword.

If you must refer to a non-standard identifier in a SQL statement, surround the identifier with any of the following escape characters:

Technique	Example
Double quotes	"column name"
Escaped double quotes	\"column name\"
Square brackets	[column name]
Back ticks (accent grave)	`column name`

Column Data Types

All data is stored in tables within the ActiveSpaces data grid. The columns of a table can be defined by using the following data types only:

- `long`
- `double`
- `string`
- `datetime`
- `opaque`

When using table operations to store data into the data grid, the `Row` API provides methods for setting each of these data types into the columns of a row before sending that row to the data grid.

When using SQL statements, for information on how these data types map to SQL data types, see the section "SQL Data Type Mapping" in *TIBCO ActiveSpaces® Administration*.

tibDateTime

Date and time information can be stored in ActiveSpaces as a `tibDateTime` type as an alternative to storing a date as a `string`, `long`, or `double` value. `tibDateTime` consists of two 64-bit integers; one for the number of seconds since January 1, 1970 (Unix epoch), and one for the number of nanoseconds *after* the time that the *sec* component denotes.

Note: ActiveSpaces does not support time zone information as a part of date and time values. In ActiveSpaces, `tibDateTime` values are UTC time. When inserting a date or time string into a column defined with `datetime` data type, an error occurs if a non-UTC time zone is used.

Using tibDateTime Columns with tibdg and the Client API

ActiveSpaces tables can be defined by using `tibdg` with columns of the `datetime` type that are mapped internally to a `tibDateTime` object for storing data into rows of the table as shown in the following example:

```
tibdg table create mytable empid long
column create mytable startdate datetime
```

Columns that are mapped internally to `tibDateTime` can also be used as primary keys or secondary indexes. For example, to define a secondary index using the `startdate` column created in the previous `tibdg` command, use the following command:

```
tibdg index create mytable myindex startdate
```

For more information on using tibdg to define tables and indexes, see the topic "Defining a Table" in *TIBCO ActiveSpaces® Administration*.

After the table and indexes are defined, you can use the ActiveSpaces Client API to store values into and retrieve values from columns defined as `datetime`. For example, to use the Client API to store values into a row of a table, use the following C API function:

```
void tibdgRow_SetDateTime(tibEx e, tibdgRow row, const char *columnName,
const tibDateTime *value)
```

To use the Client API to retrieve values from a row of a table, use the following C API function:

```
tibDateTime* tibdgRow_GetDateTime(tibEx e, tibdgRow row, const char*
columnName)
```

Populating tibDateTime

Windows and Unix platforms use different structures for retrieving date/time data. The following are examples of how you can get the current date and time on each platform and populate a `tibDateTime` object:

Unix platforms

```
#include <sys/types.h>
#include <sys/time.h>
#include “tibdg/tibdg.h”

struct timeval    timebuffer;
(void) gettimeofday(&timebuffer, NULL);

tibDateTime dt;
dt.sec  = timebuffer.tv_sec;
dt.nsec = timebuffer.tv_usec * 1000;
```

Windows

```
#include <sys/timeb.h>
#include “tibdg/tibdg.h”

truct __timeb64 timebuffer;
ftime64_s(&timebuffer);

ibDateTime dt;
t.sec  = timebuffer.time;
dt.nsec = timebuffer.millitm * 1000000;
```

The ActiveSpaces API can then be used to store the `tibDateTime` object into a row that is then stored into a table in the data grid.

Using tibDateTime Columns with SQL Commands

You can use tibDateTime columns with SQL commands.

The SQL CREATE TABLE command can be used to define a table with columns that map internally to a `tibDateTime` object by using the following SQL data types:

- DATETIME
- DATE
- TIME
- TIMESTAMP

The following SQL CREATE TABLE command is an example of creating a table with a column of DATETIME type:

```
CREATE TABLE IF NOT EXISTS mytable (empid BIGINT PRIMARY KEY, startdate
DATETIME)
```

The following SQL CREATE INDEX command is an example of defining a secondary index for a table that includes a DATETIME column:

```
CREATE INDEX IF NOT EXISTS myindex ON mytable (startdate)
```

Populating tibDateTime Columns

After you have created a table and defined its indexes, you can store data into a DATETIME column as follows:

```
INSERT INTO mytable (empid, startdate) VALUES (1, '2020-09-20
00:00:00.000000000Z')
```

`YYYY-DD-MM HH:MM:SS.SSSSSSSSSZ` is the default format of a date or time string that is used with SQL commands.

To facilitate inserting data, ActiveSpaces also provides support for a subset of ISO 8601 date or time strings. These strings are in the following format:

```
  YYYY-MM-DD[Thh.mm.ss[.s][TZD]]
```

where

T is used to delimit the date from the time.

.s indicates fractional seconds and can be up to nine digits long.

A "." by itself is not valid.

TZD is the time zone designator.

ActiveSpaces supports storing time only in UTC, so TZD can be one of the following values:

- Z
- +00:00
- +0000
- +00

A space is not allowed between the time and TZD.

Therefore, the following INSERT statement is equivalent to the previous one mentioned in this topic:

```
INSERT INTO mytable (empid, startdate) VALUES (1, '2020-09-20')
```

Query tibDateTime Columns

Data stored in tibDateTime columns can be queried using a SQL SELECT statement as follows:

```
SELECT * FROM mytable WHERE startdate='2020-09-20 00:00:00.000000000Z'
```

When querying tibDateTime columns, you must always use the default date or time string format. The only exception to this rule is when you are trying to directly retrieve a single row of a table by using only its primary key. In such a case, you can use a date/time string in the ISO 8601 format mentioned in the tibDateTime section. For example, if you had defined your table using the following command:

```
CREATE TABLE mytable (empid BIGINT, startdate DATETIME, PRIMARY KEY
(empid, startdate))
```

then in the WHERE clause of your SELECT statement, you can use ISO 8601 formatted date/time string as shown in the following example:

```
SELECT * FROM mytable WHERE empid=1 AND startdate='2020-09-20'
```

or

```
SELECT * FROM mytable WHERE empid=1 AND startdate='2020-09-
20T00:00:00+00:00'
```

Querying a Data Grid Table

Application programs have two options available for querying the data in the data grid:

- Table iterator
- `Session` statement

A table iterator is useful for basic queries on a table where all of the columns of a table are expected in each row of the query results.

A `Session` statement is used to execute SQL SELECT statements. Therefore, you have more control over the columns in the query results and also on whether the results must be aggregated or sorted. Using the statement, you can optionally set parameters in a WHERE clause as opposed to specifying an entire string for the filter in a table iterator.

Table Iterator

A table iterator is used when you have created a table object and you have to iterate over all the rows or a specific subset of the rows in the table.

You can create a table iterator to query the contents of the table and then iterate over the query results. Using a NULL filter string when creating the table iterator returns all of the rows of the table. Providing a non-NULL filter string when creating a table iterator controls the rows of a table that are included in the query results for the iterator.

The filter string format follows the syntax of the WHERE clause of a SQL SELECT statement excluding the WHERE keyword as shown in the following example:

```
column_name > 100
```

When applied to a row of the table, the filter string results in a boolean value indicating whether or not a row must be included in the results of a query. See The WHERE Clause.

The results of a query using a table iterator contain all columns of a table rather than a subset of specific columns. That is, all queries using a table iterator implicitly begin with `SELECT * FROM table_name WHERE`. Nonetheless, programs do not specify this string, they specify only the filter that would follow the WHERE keyword.

Application programs cannot influence the order of the results.

Session Statement

A `Statement` is created from a `Session` object and is not tied to a particular table object. A `Session` statement is created by using a SQL SELECT string and the table for the query is determined by parsing the SELECT string.

The SQL SELECT string supported has the form:

```
SELECT <select list> <from clause> [<where clause>] [<group by clause>]
[<order by clause>] [<limit clause>]
```

For a complete description of each component of the SELECT syntax, see the subsections under The SQL SELECT Statement.

For more information about statements, see Statement.

Advantages of a Session Statement over a Table Iterator

Using a `Session` statement to execute SQL SELECT statements has the following advantages over using a table iterator:

- You do not have to create a `Table` object before you can query a table.
- The same query can be run multiple times using a single `Statement` object.
- To aid with data security, you can use parameters to decouple the query from the data values used in the query.
- Parameters can be used to vary the result of your query each time it is run.
- You can specify a subset of the columns of a table to be returned in the rows of the query results.
- Aggregation functions can be applied to the query results.

- A GROUP BY clause can be used to aggregate the results of a query.
- An ORDER BY clause can be used to sort the results of a query.
- A LIMIT clause can be used to set an upper bound to the number of rows a query can return.

Data Consistency for Queries

When querying a table, the query processes the data in the table at the time the query is initiated.

As the table data is distributed across the nodes in a data grid, ActiveSpaces provides properties that affect whether or not the table data used for the query is consistent, with respect to partially committed transactions, across all nodes of the data grid. There are two data consistency levels that can be applied to queries:

Global Snapshot (default level)

Data retrieval is synchronized across copysets to ensure that the data accessed by the query is from committed transactions.

Snapshot

This consistency level makes no guarantee that the data accessed by the query is not from partially committed transactions. With snapshot level consistency it is possible that data from a committed transaction has not been written to all copysets when the query executes. Therefore, the rows of the query result can contain data from a partially committed transaction.

For both table iterators and session statements, you can specify a consistency property when the objects are first created. If a consistency property is not specified, the default consistency level of global snapshot is used.

For statements, you can override the consistency level each time the query is run by specifying a consistency property when the method to execute the query is invoked.

Query snapshots are inexpensive if the table data changes slowly, but can become expensive if the data changes rapidly. To limit memory growth within data grid components, administrators can limit the number of concurrent snapshots.

Regardless of the data consistency level used, rows returned for a query are not locked by the session.

For more information about setting statement properties, see Properties.

For more information about setting table iterator properties, see Table Iterator.

The SQL SELECT Statement

A SQL SELECT statement is used to query data in the data grid. The table to query is determined by parsing the SELECT string when creating the Statement object. The rows that satisfy the query are returned in a ResultSet. A WHERE clause can be used in the SELECT statement to control which rows of a table must be used in the query result. An ORDER BY clause can be appended after the WHERE clause to sort the resulting rows of the query. A LIMIT clause can be appended as the last clause of the SELECT string to control the number of rows ultimately returned by the query.

SQL keywords, table, and column identifiers are not treated as case sensitive when used in a SQL SELECT statement. However, string values are treated as case sensitive and must be surrounded by single quotes.

The Syntax of a SELECT Statement

The SELECT syntax supported has the following format:

```
SELECT <select list> <FROM clause> [<WHERE clause>] [<GROUP BY clause>]
[<ORDER BY clause>] [<LIMIT clause>]
```

Notice that <FROM clause> is not optional. Use of a SELECT statement with ActiveSpaces is always intended to query data in the data grid and the table to query must always be specified by including <FROM clause>.

The order of rows returned for a query is non-deterministic unless an ORDER BY clause is included in the SELECT statement.

Unsupported SQL Features

ActiveSpaces does not support the following SQL features:

- GLOB operator
- UNIQUE
- EXISTS
- ALL
- DISTINCT (except with GROUP_CONCAT function)

- HAVING
- Nested queries
- Joins
- Window functions

The FROM Clause

The FROM clause specifies the table to query. When using ActiveSpaces, a FROM clause is required in a SELECT statement. The FROM clause syntax supported by ActiveSpaces is the following:

```
FROM <table name> [ [AS] <correlation name> ]
```

A correlation name is an identifier that is associated with the specified table and can be used in place of the table name anywhere within the SELECT statement. The following code snippet serves as an example:

```
SELECT t.* FROM mytable AS t WHERE t.col1 = 100
```

The Select List

The select list specifies the columns of the rows in the query result. These columns are not necessarily columns from a table, but may be columns whose value is derived by applying a function or equation to the rows of the table selected for the query.

ActiveSpaces uses the following select list syntax:

```
[<identifier>.]<asterisk> | <value expression> [ [AS] <label> ]
```

Asterisk

An asterisk, '*', used in the select list refers to all columns of the table that is specified in the FROM clause. Each column of the table is included exactly once in each row of the query results.

To select all the columns of a table or correlation, use ".*' (dot asterisk) as the suffix as shown in the following example:

```
SELECT mytable.* FROM mytable
```

When an asterisk is used in the select list, the names of the columns as defined for the table are used as the label when accessing the columns in the rows of the query result.

Value Expression

A value expression specifies the value returned by a particular column for each row of the query results.

A value expression can consist of the following items:

- Column Identifiers
- Functions
- Literals (For example, 1234, 'somestring')
- Expressions (For example, x+y, x AND y)

A value expression can be given a label to use when accessing the data for each column in a row of the query results as shown in the following example:

```
SELECT col1 AS c1 FROM mytable
```

If a value expression is not given a label, the entire string used to specify the value expression is used as the label as shown in the following example:

```
SELECT col1, col2 + col3, ‘row totals’ FROM mytable
```

To access the columns in the rows of the query result for the SELECT statement above, the following labels would be used:

- 'col1'
- 'col2 + col3'
- 'row totals'

For more information about expressions, see SQL Expressions.

Restrictions on Using a Value Expression

A value expression can be composed of any of the items described in Value Expression, but with the following restrictions:

- Functions must be supported by ActiveSpaces.
- A value expression cannot contain an aggregate function.
- A select list that contains an aggregate function cannot also contain a value expression.
- A value expression cannot contain parameters.
- A value expression cannot contain SELECT statements.
- Nested expressions are limited to a depth of 100.

Expressions are parsed into a tree for processing. If you reach the expression depth limit, consider using parentheses to reduce the depth of your expression

For information about the limitations of using value expressions that are aggregate functions, see the section on "Aggregate Functions" in Functions Used in The Select List .

CASE Expressions used in the Select List

You can use CASE expressions in the select list of SELECT statements.

For more details on CASE Expressions, see CASE Expressions.

Using a CASE expression in the select list of a SELECT statement can either help categorize results based on their values or help apply different calculations to a result based on its value. For example, the following query can be used to find out the number of employees that are in 'in-state' versus 'out-of-state':

```
SELECT empid, CASE state WHEN ‘IL’ THEN ‘in-state’ ELSE ‘out-of-state’
END AS location FROM employee
```

Case Expression Restrictions

In addition to the restrictions on value expressions listed in Restrictions on Using a Value Expression, the use of a CASE expression in a select list has the following additional restrictions:

- CASE expressions cannot contain functions that use the DISTINCT keyword.
- All resulting values of the CASE expression must be of the same data type. The resulting data types are not evaluated until the query is executed. Therefore, it is possible that a bad SELECT statement can be created, which later causes an exception to be returned after executing the query.
- CASE expressions that contain subexpressions are parsed, but do not return an error.

Additionally, an internal limit is applied to nested CASE expressions to prevent the exhaustion of resources. This limit is different from the expression depth limit and can vary based on the composition of the query. The limit is applied during the parsing of the nested CASE expression. When the limit is reached, the following error is returned:

```
Error Code  = Resource limit reached
Description = SQL parser stack overflow
```

Functions Used in The Select List

ActiveSpaces supports the use of aggregate functions, date and time functions, and string functions as a value expression or as part of a value expression.

When used as the value expression, the result of the function is used as the value for columns in the select list.

Date and Time Functions

Date and time function usage is supported for value expressions as shown in the following example:

```
SELECT date(dtmcol) FROM mytable
```

For more information on date and time functions, see Date and Time Functions.

When using a timevalue of now with a date and time function in a value expression, now is converted to the current date and time by each node of the data grid as the query is executed and rows are found to use in the query results. Carefully analyze your use of now to ensure the query returns the results you expect. The same holds true for using the SQL variables CURRENT_DATE CURRENT_TIME and CURRENT_TIMESTAMP in a value expression.

String Functions

String functions can be used in value expressions. For example, to return the `id` column's value and the uppercase value of the `lastname` column from all rows of a table you can use the following query:

```
SELECT id, UPPER(lastname) FROM mytable
```

For more information on the string functions supported by ActiveSpaces, see SQL String Functions.

Aggregate Functions

Aggregate functions as value expressions use the column values from multiple rows to calculate a single value. Aggregate functions are applied to column values of the rows selected to be used for the query results.

Depending on the syntax used for the SELECT statement, these rows can be the result of the GROUP BY clause or the WHERE clause. Rows from the GROUP BY clause take precedence over rows from the WHERE clause. If both the GROUP BY and WHERE clauses are omitted, values from all rows of the table are used for the aggregation.

Aggregate functions ignore NULL values. If all the values are NULL or cannot be converted to the appropriate data type for the function, the result of the function is NULL. The aggregate functions supported are listed in the following table:

Function	Description
`AVG(<column>)`	Computes the arithmetic mean of the non-NULL values in the column. The type of the result is always a Double, regardless of the data type of the underlying column. If the data type of the underlying column is String, the value is converted to a Double. If the data type of the underlying column is Opaque, the data is treated as a String and then converted to a Double.
`COUNT(<column>) COUNT(*)`	Computes the number of non-NULL values in the column, or the number of rows. The result is always of type Long. **Note: Note:** `COUNT()` is not supported.

Function	Description
`MIN(<column>)`	The minimum non-NULL value in the column. The type of the result matches the type of the underlying column.
`MAX(<column>)`	The maximum non-NULL value in the column. The type of the result matches the type of the underlying column.
`SUM(<column>)`	Computes the sum of the non-NULL values in the column. The type of the result is Long if the data type of the underlying column is Long, otherwise it is Double. If the data type of the underlying column is String, the value is converted to a Double. If the data type of the underlying column is Opaque, the data is treated as a String and then converted to a Double.
`GROUP_CONCAT( DISTINCT <column>)`	Concatenates the unique values for a column in the group into a comma separated list. The DISTINCT keyword is required. The order of the values in the list is not defined. The type of the result is always a String, irrespective of the data type of the underlying column. If the data type of the underlying column is Long or Double, the value is converted to a String. If the data type of the underlying column is Opaque, the data is treated as String.

For more information about the aggregate functions supported by ActiveSpaces, see Aggregate Functions.

The WHERE Clause

A WHERE clause is used to determine whether a row of a table must be used when composing the results of a query.

When applied to the row of a table, the WHERE clause must result in a boolean value indicating whether or not a row must be used when calculating the results of a query. The WHERE clause syntax supported by ActiveSpaces is the following:

WHERE [NOT] <predicate> [AND | OR [NOT] <predicate>] . . .

A predicate is a condition expression that evaluates to true or false. If the predicate evaluates to true for a row, that row is used when calculating the results of the query. For

example:

```
city='Chicago'
percent<=75.0
```

The AND and OR operators are used to filter rows based on more than one condition. For example:

```
city='Chicago' AND lastname='Dailey'
```

The NOT operator includes a row if the condition is not true. For example:

```
NOT city='Chicago'
```

When you create an SQL statement, the efficiency of the SQL statement depends on how well you construct a WHERE clause. For example, a WHERE clause must be used to prevent a full table scan from being done. For more information about constructing an efficient WHERE clause, see Tips on Constructing an Efficient WHERE Clause.

Tips on Constructing an Efficient WHERE Clause

When you create a SQL statement, it is important to construct an efficient WHERE clause.

Indexes and WHERE Clauses

The performance of a SQL statement that includes a WHERE clause depends partly on the way you construct your WHERE clause and partly on the definition of table indexes by the data grid administrator. In general, you want to construct a WHERE clause that results in the primary key or a secondary index to be selected for evaluating the WHERE clause predicates against a subset of the rows of a table.

Omitting the WHERE clause or using a WHERE clause that results in no key or index being selected might end up in a full table scan. Full table scans iterate over every row in a table to see if the row must be used by the SQL statement thereby making them inefficent. Therefore, TIBCO recommends that you avoid using full table scans.

For example, if the primary key for a table is defined on the `empid` column, the following query can directly access one row of the table:

```
SELECT * FROM mytable WHERE empid='ID-1234'
```

If there are no secondary indexes defined for the table, the following query performs a full

table scan:

```
SELECT * FROM mytable WHERE lastname LIKE ‘B%’
```

However, if there is a secondary index defined on the `lastname` column, then the query above would reduce the number of rows scanned to only those rows that begin with a 'B' or 'b'.

Programmers: consult your data grid administrator for information about the definition of indexes.

Programmers and administrators can use the rules of thumb in the following sections to help promote efficiency and high performance of queries.

Primary Key and Secondary Indexes

Rule of Thumb: Construct WHERE clauses in which every predicate refers to columns of the primary key or a secondary index.

A WHERE Clause is composed of one or more predicates. A predicate is a condition expression that evaluates to either true or false. Each predicate must be composed of columns from the primary key or a secondary index. This allows the primary key or a secondary index to be selected to find the rows of the table to use for evaluating the predicate. If a primary key or secondary index cannot be selected for even one predicate in the WHERE clause, a full table scan is performed even if the primary key or a secondary index is selected for all of the other predicates in the WHERE clause.

Left-Most Columns

Rule of Thumb: Construct WHERE clauses that reference the left-most columns of a primary key or secondary index using the operators =, ==, <=, >=, <, >, IN, IS, or BETWEEN.

Not all columns of the primary key or secondary indexes have to be referenced by a WHERE clause predicate. When a primary key or secondary index includes more than one column, the administrator has defined them from left to right. For the primary key or a secondary index to be considered for use in evaluating a predicate, at least the left-most column of the key or index must be referenced by the predicate. The primary key or a secondary index is not selected, if the WHERE clause skips the left-most column but refers only to columns defined further to the right.

Similarly WHERE clauses that refer to the left-most two columns can be even more efficient. Queries can achieve maximum efficiency when they use WHERE clauses that refer to all of the columns of the primary key or a secondary index.

When a WHERE clause does not contain the left-most column of the primary key or any of the secondary indexes, a full table scan results. A query that does a full table scan is least efficient and must be avoided.

The order in which columns appear in the WHERE clause does not affect query efficiency. Only the order of columns when defining the index matters.

Avoid Left-Most NOT

Rule of Thumb: Do not construct WHERE clauses that reference the left-most columns of the primary key or secondary index using the operators NOT, IS NOT, !=, <>, ISNULL, IS NULL, NOTNULL, NOT NULL, and IS NOT NULL.

In contrast to the rule of left-most columns, a *WHERE clause* that references the left-most columns with these operators have the opposite effect: to guarantee a full table scan, which is the least efficient.

This rule does not imply that operators in the NOT family are always inefficient. For example, a query can still be efficient if it obeys the left-most columns rule and also tests columns further to the right using NOT. For example, if the administrator defined an index on the columns `lastname` and `firstname`, then this WHERE clause can be efficient:

```
WHERE lastname='Smith' AND firstname IS NOT
'Dan'
```

Limit Range Queries From Both Ends

Rule of Thumb: When using the operators > or >=, which specify a lower limit on a column's value, also include the opposite operators, < or <=, to specify an upper limit on the same column.

A query searches an index from its lower limit to its upper limit. If you omit the upper limit, the query continues searching to the end of the index. If you omit the lower limit, the query begins with the first row of the index.

Operators Used in the WHERE Clause

A predicate can use the operators as described in Operators. For detailed information about using the LIKE operator, see LIKE Operator.

The following sections contain additional information about using the LIKE operator in the predicate of a WHERE clause.

Indexed Columns with LIKE Operator

If the left operand of the LIKE operator is the name of an indexed column of type string, ActiveSpaces converts the LIKE operator into a range query using >= and <. This enables the ActiveSpaces index selection algorithm to select the index and use it for scanning rows when processing the query. For example, take a look at the following statement:

```
SELECT * FROM mytable WHERE lastname LIKE ‘Long’
```

Internally this statement is converted to the following statement and the index on `lastname` can be used to perform a range scan.

```
SELECT * FROM mytable WHERE lastname >= ‘Long’ AND lastname < ‘long’
```

However, the index for `lastname` cannot be used if `<character sequence pattern>` is a `long` value or starts with a wildcard (%, _) or digit.

Pattern Matching with LIKE Operator

When using the LIKE operator in a WHERE clause, using a character sequence pattern that begins with a wild card results in a full table scan.

It is also important to be aware that you might end up scanning more rows than you would expect because of the way pattern matching works when matching is not case sensitive. For example, if you have an index defined for `lastname` and you run the following query:

```
SELECT * FROM mytable WHERE lastname LIKE
'm%'
```

The index is used, but all the rows are scanned where the value of `lastname` start with "M" through "Z" and "a" through "m".

Using Date and Time Functions in the WHERE Clause

Date and time information can be stored in the data grid as date and time strings, Julian Day long or double values, or as tibDateTime values.

Date and time functions can be used in the WHERE clause for querying these different types of columns as shown in the following example:

```
SELECT * FROM mytable WHERE date(dtmcol)=’2016-12-24’
```

For more information about date and time functions, see Date and Time Functions.

Function Usage on Index Columns

When using date and time functions on primary key or secondary index columns in a WHERE clause or table iterator filter `string`, the application of the function to the column, in the left side of an equation, does not reduce the number of rows being examined for a query. Indexes are used based on comparisons to the actual value of a column and not the value that is the result of conversion by functions.

You can use the date and time functions in comparisons with key or index fields and still get the benefit of the key or index, if you use the function on the right-hand side of the equation. For example, if the `dtm` column is of type `datetime` (holds `tibDateTime` objects) and `dtm` is defined as a secondary index, the following query only scans those rows with a date less than '2018-06-01 00:00:00':

```
SELECT * FROM mytable WHERE dtm < datetime('2018-06-01')
```

WHERE Clause Examples

Using the `date()` function always returns a date of the form:

- YYYY-MM-DD

Where:

YYYY [0000-9999]

MM [01-12]

DD [01-31]

Given a table with the following columns and values for 2016-12-24 00:00:00:

Columns	Data Type	Values
`dtm`	`datetime`	[1482566400, 0]
`dtmstr`	`string`	“2016-12-24T00:00:00”
`dtmlong`	`long`	1482566400
`julianday`	`double`	2457746.50000

Assuming all columns of a row are loaded with a date or time value that contains the date 2016-01-01, the following are the examples of using the `Date()` function in a WHERE

clause:

```
SELECT * FROM mytable WHERE date(dtm)='2016-12-24'
SELECT * FROM mytable WHERE date(dtmstr)='2016-12-24'
SELECT * FROM mytable WHERE date(dtmlong,'unixepoch')='2016-12-24'
SELECT * FROM mytable WHERE date(julianday)='2016-12-24'
```

Querying tibDateTime Columns

To perform a SQL query on a `tibDateTime` column, the value of the `tibDateTime` column is internally converted to a string of the form:

```
YYYY-MM-DD HH:MM:SS.SSSSSSSSSZ
```

Where the ' SSSSSSSSS ' following ' SS. ' represent nanoseconds and 'Z' represents Coordinated Universal Time (UTC).

ActiveSpaces internally converts `tibDateTime` column values to strings in this format. As a result, the SQL date and time functions, which normally only work on ISO 8601 formatted strings can also be used on the converted `tibDateTime` values. The date and time functions are lenient with respect to the number of digits following the decimal in the fractional seconds, so the nine places of precision in the string representation of a `tibDateTime` value does not cause any issues.

The following are examples of queries, which can be run on a column of type `datetime` that contains a `tibDateTime` object:

```
SELECT * FROM mytable WHERE dtm='2016-01-01 00:00:00.000000000Z'
SELECT * FROM mytable WHERE date(dtm)='2016-01-01'
SELECT * FROM mytable WHERE datetime(dtm)='2016-01-01 00:00:00'
```

For more information about using the SQL SELECT statement to query `tibDateTime` columns, see tibDateTime.

Error Conditions

Using a date or time function on a column that does not contain an appropriate date or time value does not cause an exception. In most cases, a SQL NULL is the result of the function and the query is processed accordingly.

Using String Functions in the WHERE Clause

String functions can be used in WHERE clause predicates.

For more information about string functions, see SQL String Functions.

Using Multi-Argument MIN, MAX Functions in the WHERE Clause

When the functions `MIN()` and `MAX()` are used in a WHERE clause, the multi-argument forms of these functions must be used.

In a *WHERE clause*, the following functions do not perform aggregation as they would if they were used in a select list:

- `MAX(x, y, …)`: The multi-argument MAX function returns the argument with the maximum value.
- `MIN(x, y, …)`: The multi-argument MIN function returns the argument with the minimum value.

The GROUP BY Clause

A GROUP BY clause is used to arrange rows with identical values in columns into groups.

The GROUP BY clause immediately follows the WHERE clause of a SQL SELECT statement. For more information about the syntax of a SQL SELECT statement, see The Syntax of a SELECT Statement. The following code snippet is an example of the syntax of the GROUP BY clause:

```
GROUP BY <grouping term>[, <grouping term>] . . .
```

A grouping term is a reference to a value expression from the select list. The following query is invalid because state is not in the select list:

```
SELECT AVG(salary) FROM employees GROUP BY state
```

A GROUP BY clause is most often used with aggregation functions in the select list to provide summary information about each group of rows. For example, the following query returns a set of rows - one for each state, with each row containing two columns, the state and the mean salary for employees in that state:

```
SELECT state, AVG(salary) FROM employees GROUP BY state
```

A value expression, which is not an aggregate function, must normally be included as a grouping term in the GROUP BY clause. However, this is not a requirement.

Limitations

Currently, the GROUP BY clause can only be applied to value expressions that are column identifiers. Expressions and functions in the select list cannot be used as grouping terms. For example, the following query is invalid as it refers to the label of a function in the select list:

```
SELECT date(dtm) AS Year, COUNT(dtm) FROM mytable GROUP BY Year
```

Performance

When using a GROUP BY clause, the performance of the aggregation depends on how many rows are used in the aggregation. A query that includes aggregation but does not contain a WHERE clause performs aggregation using all the rows from the table and must be avoided, if possible.

For the best aggregation performance, the column used as an argument to your GROUP BY clause must have an index defined that uses that column. Your query must include a WHERE clause that causes the index to be selected for the query. For example, if you have a secondary index named `lastname_idx` defined for the `lastname` column, the following queries use `lastname_idx` to reduce the number of rows in the query on which aggregation is performed:

```
SELECT lastname FROM mytable WHERE lastname LIKE 'B%' GROUP BY lastname
```

For more information about query performance, see Tips on Constructing an Efficient WHERE Clause.

The ORDER BY Clause

An ORDER BY clause is used to sort the results for the query.

The ORDER BY clause follows both the WHERE clause and GROUP BY clause, if present.

For more information about the syntax of the SELECT statement, see The Syntax of a SELECT Statement. The syntax of the ORDER BY clause is:

```
ORDER BY <ordering term> [ASC | DESC][, <ordering term> [ASC | DESC] ]
...
```

An ordering term can be a column in the select list, an alias specified in the select list, or a column index number in the select list. Each ordering term can optionally be followed by ASC or DESC. ASC indicates that the query results must be sorted on the ordering term column in ascending order. DESC indicates that the query results must be sorted on the ordering term column in descending order. The default sort order is ascending.

TIBCO recommends using columns in the ORDER BY clause, which are columns of a primary or secondary index defined for the table. Using columns of a primary or secondary index increases the efficiency and speed of the query. ORDER BY clauses with columns that match the order of defined indexes exactly or match the order of defined indexes in the reverse order are usually more efficient than other types of ORDER BY clauses.

When the SELECT statement includes a WHERE clause, the ordering terms used in the ORDER BY clause must match the columns of the primary or secondary index selected for processing the WHERE clause to achieve the best performance.

If you do not use columns from a primary or secondary index in the ORDER BY clause, there is a per-query limit on the amount of allowed memory by default (128MB) that can be used to buffer results when attempting to order them.

Example

For example, there is an index defined for the table by the name of `index1` that is indexed on `lastname`, `firstname`, and `city`. The following queries use `index1` because the columns match the index column in the same order and the sorting is in a single direction (either in the ascending order or in the descending order):

```
SELECT * FROM t1 ORDER BY lastname ASC, firstname ASC, city ASC

SELECT * FROM t1 ORDER BY lastname DESC, firstname DESC, city DESC
```

The following query cannot use `index1` because some columns are sorted in the ascending order and some in the descending order:

```
SELECT * FROM t1 ORDER BY lastname DESC, firstname ASC, city ASC

SELECT * FROM t1 ORDER BY lastname ASC, firstname DESC, city ASC
```

LIMIT Clause

The LIMIT clause is used to specify the maximum number of rows to return for a query.

The following code shows the syntax of the LIMIT clause supported by ActiveSpaces:

```
LIMIT <rowcount expression>
```

The LIMIT clause affects the number of result rows for the query. The LIMIT clause, if specified, must be the last clause of the SELECT statement and is the last clause applied to a query. In other words, the LIMIT clause is applied after the WHERE clause is applied, the GROUP BY clause is applied, and the ORDER BY clause is applied. The LIMIT clause does not affect the number of rows scanned for a query.

LIMIT Clause and Full Table Scan Property

When the WHERE clause of a SELECT statement does not refer to the primary key or secondary indexes of a table, the entire table must be scanned to find the rows of the table that is returned for a query. This process of scanning the entire table is known as a `full table scan`. When a LIMIT clause is appended to a SELECT statement, it does not affect whether or not a full table scan is performed. However, the LIMIT clause affects the number of results returned for the query, but not how many rows are scanned for a table. The rows resulting from the scan might then also have to be grouped with a GROUP BY clause and then sorted with ORDER BY clauses. The final set of result rows are then limited by the count specified for the LIMIT clause.

Modifying Data in a Table

Application programs have the following options to add, update or delete data in tables in a data grid:

- Table PUT operation: PUT
- Table UPDATE operation: UPDATE
- Table DELETE operation: DELETE

- SQL INSERT: The SQL INSERT Statement
- SQL INSERT OR REPLACE: The INSERT OR REPLACE Statement
- SQL DELETE: The SQL DELETE Statement

The SQL INSERT Statement

A SQL INSERT statement writes new rows of data into a table. If the INSERT activity is successful, it returns the number of rows inserted into the table. If the row already exists, it returns an error. Multiple rows can be inserted into a table. Multi-row inserts treat the insertion of rows equivalent to being in a transaction whether or not a transacted session is used. If the INSERT statement is embedded in a transaction or if you are trying to insert multiple rows, the failure to insert a row results in a rolled back transaction.

Syntax

```
INSERT INTO <table_name> [(<column_name_list>)] VALUES (<column_value_
list>)[,(<column_value_list>)]...
```

where

- `table_name`: Name of the table in which the rows are inserted.
- `column_name_list`: Each column name in `column_name_list` that identifies a column of `table_name`.
- `VALUES`: Values for the columns in `column_name_list`.
- `column_value_list`: A comma separated list of values.

The following table shows the syntax for inserting rows in a table:

Number of Rows	Syntax
Single Row Insert	`INSERT INTO <table_name> [(column1 [, column2, column3 …])] VALUES (value1 [, value2, value3 …])`

Number of Rows	Syntax

Multi-Row Insert	`INSERT INTO <table_name>  [(column-a, [column-b, …])]`
	`VALUES (‘value-1a’, [‘value-1b’, …]),`
	`(‘value-2a’, [‘value-2b’, …]),`
	`. . .`

Rules for the column_name_list

- Each of the named columns of the new row is populated with the results only after evaluating the corresponding `VALUES` expression.
- If `column_name_list` is omitted, the number of values inserted into each row must be the same as the number of columns in the table.
    - Values are populated into the row in the order the columns were defined for in the table.
- All columns in `column_name_list` must be writable.
- No column name can be listed more than once.
- Not all columns of a table have to be listed. The value of any unlisted column is `NULL` (empty).
    - If a value is not provided for a primary key column, an error is reported.

Rules for the column_value_list

- The data type of each value must match the data type of the column as configured for the table.
- If `column_name_list` is given, the number of values must match the number of columns in `column_name_list`.
- If `column_name_list` is given, the n^{th} column in `column_name_list` is assigned the n^{th} value of `column_value_list`.
- If `column_name_list` is omitted, the number of values must match the number of columns in the table with an implied ascending sequence of the ordinal positions of columns in the table.
- A value can be the result of an expression.

- A value can be a parameter marker that requires a value to be bound to the parameter before the statement is run.
    - The maximum length of a SQL statement is `1,000,000` bytes.
    - Parameters must be used to insert data that can cause the statement to reach the maximum length.
    - The parameter type must match the column type exactly. Call the appropriate `setParameter()` method - the type of the parameter passed to this method must match with the column type.

For more information about using the SQL INSERT statement to store values in a `tibDateTime` column, see tibDateTime.

Adding Rows to a Table by Using the SQL INSERT Statement

Procedure

1. Formulate a SQL INSERT string for adding rows of data to a table.
2. Call the `createStatement()` method of the `Session` object. Pass the SQL INSERT string as an argument.
3. If the SQL INSERT string contains parameter markers, call the `Statement` methods to set the parameter values.
4. Run the INSERT statement by calling the `executeUpdate()` method of the `Statement` object.
5. Check the result of the `executeUpdate()` method to verify that the correct number of rows have been inserted into the table.
6. Set different parameter values and rerun the INSERT statement by repeating steps 3 to 5.
7. Close the `Statement` object.

Errors for INSERT Statement

By default, when an error occurs for an INSERT statement, changes made by the statement are undone. If the INSERT statement is called from within an existing transaction, the entire transaction is rolled back. ActiveSpaces does not impose a limit to the number of

rows in a multi-row insertion. However, the number of rows must be processed within the time specified by the `client request timeout` property of the data grid (The default is `5` seconds).

If the insert values list does not match column list, a SQL command error is returned. Errors due to a conflict with the following constraints:

Constraints	Error Description
PRIMARY KEY	Values are not provided for all primary key columns

INSERT Statement and Expressions

INSERT statements can contain expressions for the values to be inserted for a column. For example, the following INSERT statement can be used to add a timestamp as the primary key for each row being added:

```
INSERT INTO mytable (col1, col2) VALUES (datetime('now'), 'some string')
```

Note: The expressions that can be used as values are limited and cannot contain column names that must be dynamically evaluated.

Parameter Binding with INSERT Statements

When you use parameter binding, you use "?" (question mark) instead of actual values in a SQL statement. The "?" parameter must be used in place of a value for that column and not in arbitrary expressions.

The current parameter bindings are used whenever `executeUpdate()` is invoked for an INSERT statement. Parameter bindings can also be used with multi-row inserts.

Example of parameter binding:

```
INSERT INTO mytable (col1, col2) VALUES (?,?),(?,?),(?,?)
```

The maximum number of parameters on a given INSERT statement is 999.

The INSERT OR REPLACE Statement

You can use the INSERT OR REPLACE statement to write new rows or replace existing rows in the table. The syntax and behavior of the INSERT OR REPLACE statement is similar to the INSERT statement. Unlike the INSERT statement, the INSERT OR REPLACE statement does not generate an error if a row already exists.

For information about the INSERT statement, see The SQL INSERT Statement.

Syntax

```
INSERT OR REPLACE INTO <table_name> [(<column_name_list>)] VALUES
(<column_value_list>)[,(<column_value_list>)]...
```

where

- `table_name`: Name of the table in which the rows are inserted.
- `column_name_list`: Each column name in `column_name_list` identifies a column of `table_name`.
- `column_value_list`: Values for the columns in `column_name_list`.

The SQL DELETE Statement

A SQL DELETE statement removes rows of data from a table. If the DELETE activity is successful, it returns the number of rows removed from the table.

If the DELETE activity is not successful, 0 is returned as the number of rows removed from the table.

Transactions are not supported for DELETE statements and you cannot create a DELETE statement from a transacted session.

Since transactions are not supported with DELETE, if an error occurs while performing row deletion, any rows already deleted remain deleted and the delete is not rolled back.

Syntax

```
DELETE FROM <table_name> [ AS <alias> ] [ <WHERE clause> ]
```

Where

- `table_name`: Name of the table in which the rows are removed.
- `alias`: A temporary name for the table used to make the table name more readable.
- `WHERE clause`: An expression which starts with the keyword 'WHERE' and results in a boolean value that indicates whether a row should be removed. If omitted, all rows of the table are removed. For more information, see The WHERE Clause.

> ⚠ If a WHERE clause is omitted from a DELETE statement, all rows in the table are removed. For information on how to prevent the inadvertent removal of rows by DELETE statements without a WHERE clause, see the full_table_delete table and grid properties.

Removing Rows From a Table Using the SQL DELETE Statement

Procedure

1. Formulate a SQL DELETE string for removing rows of data from a table.
2. Call the `createStatement()` method of the `Session` object. Pass the SQL DELETE string as an argument.
3. If the SQL DELETE string contains parameter markers, call the `Statement` methods to set the parameter values.
4. Run the DELETE statement by calling the `executeUpdate()` method of the `Statement` object.
5. Check the result of the `executeUpdate()` method to verify that the correct number of rows have been removed from the table.
6. Set different parameter values and rerun the DELETE statement by repeating steps 3 to 5.
7. Close the `Statement` object.

Deleting large number of rows

> ⚠ When removing a large number of rows from a table, TIBCO recommends to remove batches of rows versus the entire set of rows.

To delete a large number of rows from a table, design a SQL DELETE statement as follows:

1. Design a range query which is bounded on both ends to select a subset of rows to be removed from the table.
2. Avoid unbounded range queries to reduce the number of rows scanned. For example, use a WHERE clause similar to the following:

   WHERE key >= 0 AND key <=1000

   WHERE key BETWEEN 0 AND 1000
3. Use parameters in the range query which results in a subset of rows being removed each time `executeUpdate()` is invoked. For example,

   WHERE key BETWEEN ? AND ?
4. Design the WHERE clause so that an index is used for finding the rows and prevent full table scans.
5. Set the statement property `TIBDG_STATEMENT_PROPERTY_DOUBLE_UPDATE_TIMEOUT` so the `executeUpdate` requests do not timeout.

> The SQL LIMIT clause is not part of the syntax for SQL DELETE.

Errors for DELETE Statement

By default, when an error occurs for a DELETE statement, rows already deleted remain removed and their removal cannot be undone.

When removing a large number of rows from a large table, it is recommended to remove rows in smaller batches which can be processed before the `executeUpdate()` method times out.

Authentication for Deleting Rows

When authentication is used with an ActiveSpaces data grid, a user must have write permission for a table to be able to delete rows from a table.

Properties to Control the Removal of Rows From a Table

To help in the successful removal of rows from a table, there are several ActiveSpaces properties which can be used.

Statement Properties

Statement properties are defined when a SQL statement is first created or when a statement is executed. For more information, see Properties.

`TIBDG_STATEMENT_PROPERTY_DOUBLE_UPDATE_TIMEOUT` - When an `executeUpdate()` is invoked, it does not return until all rows satisfying the WHERE clause are removed. This property can be used to increase the client's timeout (in seconds) of an `executeUpdate()` request for a DELETE statement. This property overrides the grid's `client_req_timeout` setting.

Table Properties

Table configuration properties are specified when you define a table in the data grid. See the sections "Defining a Table" and "Defining a Table by Using SQL DDL Commands" in *TIBCO ActiveSpaces® Administration*.

The following table configuration properties can be used to control the ability to remove all rows from a table:

`full_table_delete` - When set to other than *inherited* this property can be used to override the grid's `full_table_delete` configuration setting. tibdgproxy enforces the `full_table_delete` setting. Log messages resulting from this setting are stored in the tibdgproxy logs.

Grid Properties

Grid configuration properties are specified when you define a data grid and are applied to all relevant objects in the data grid. See the section "Defining a Data Grid" in *TIBCO ActiveSpaces® Administration*.

The following grid configuration properties can be used to control the ability to remove all rows from a table for SQL DELETE statements:

- `full_table_delete` - Used to control the ability to remove all rows from any table in the data grid when a DELETE statement does not contain a WHERE clause. tibdgproxy enforces the `full_table_delete` setting. Log messages resulting from this setting are stored in the tibdgproxy logs.
- `full_table_scans` - Used to control the ability to remove all rows from any table in the data grid when a DELETE statement contains a WHERE clause but an index could not be found for the WHERE clause resulting in a full table scan being done to find the rows to delete. tibdgnode enforces the `full_table_scans` setting. Log message

results from this setting are stored in the tibdgnode logs.

The following grid configuration properties can be used to control the amount of time before a client request times out.

`client_req_timeout` - The default client request timeout is 5 seconds. When an `executeUpdate()` method is invoked, it does not return until all rows satisfying the WHERE clause are removed. If the `client_req_timeout` occurs before the `executeUpdate()` method is finished, the `executeUpdate()` method is canceled by the client. When the `executeUpdate()` for a DELETE statement is canceled, 0 is returned as the number of rows removed and any rows removed prior to the DELETE being canceled remain deleted.

`client_req_timeout` is applied to all requests from a client to the data grid. See the statement property `TIBDG_STATEMENT_PROPERTY_DOUBLE_UPDATE_TIMEOUT` for information on controlling the timeout for individual DELETE statements.

SQL Expressions

Expressions can be used in the select list and WHERE clause of SELECT statements. An expression used in a select list is called a value expression. An expression used in a WHERE clause is called a predicate.

This section contains information that is common between both types of expressions. For specific information about value expressions, see Value Expression. For specific information about predicates, see The WHERE Clause.

Operators

Operators are used in expressions to compare a column's value against another value.

Expressions can be composed with the operators from the following table:

Operator	Description
=	Tests what is on each side of the operator for equality.
==	
IS	

Operator	Description
!= <> IS NOT	Tests what is on each side of the operator for inequality.
>	
<	
>=	
<=	
ISNULL IS NULL	Tests that the row does not contain a value in this column.
NOTNULL NOT NULL IS NOT NULL	Tests that the row contains a value in this column.
BETWEENvalue_1 and value_2	Requires two values, separated by the keyword *and*. The range includes the end values.
IN(value [,value]*)	Requires a set of values, separated by commas, surrounded by parentheses.
value_1 LIKE value_2	Searches the left operand for the pattern specified by the right operand. For details about the LIKE operator, see LIKE Operator.

Value

value can be any value of the same data type as the column's data type.

LIKE Operator

The LIKE operator is used to search the left operand for a character sequence pattern specified by the right operand.

Syntax:

```
<left operand> LIKE <character sequence pattern> [ ESCAPE <char> ]
```

Two wildcard characters can be as specified in the following patterns:

- % (percent) - matches 0, 1, or multiple characters
- _ (underscore) - matches a single character

If the character sequence pattern needs to include one of the wildcard characters as one of the characters to match, you can specify an escape character to use by specifying the optional ESCAPE clause after your LIKE pattern as shown in the following example.

```
completed LIKE ‘100\%’ ESCAPE ‘\’
```

Pattern Matching

The pattern matching is not case sensitive for upper or lowercase ASCII characters. For example:

```
lastname LIKE ‘long’
```

searches the `lastname` columns for values of LONG, lONG, loNG, lonG, long, LoNG, LonG, and so on.

The pattern matching is case sensitive for Unicode characters that are beyond the ASCII range. For example:

```
lastname LIKE ‘Ünder’
```

searches the *lastname* column for values of ÜNDER, ÜNDEr, ÜNDer, ÜNder, Ünder. It does not search for `lastname` column values that start with üNDER, üNDEr, and so on.

Pattern matching that is not case sensitive works according to the order of characters in the ASCII table. Uppercase characters sort before lowercase characters. In the ASCII table, A=65 and a=97. Therefore, 'A' sorts before 'a'.

The following table contains examples of character sequence patterns.

Pattern	Finds Values That...
'a%'	start with 'a' or 'A'

Pattern	Finds Values That...
'%a'	end with 'a' or 'A'
'a%o'	start with 'a' or 'A' and end in 'o' or 'O'
'a_'	start with 'a' or 'A' and are two characters long
'_a%'	have 'a' or 'A' as the second character
'%at%'	contain 'AT', 'At', or 'at'
'a_%_%'	start with 'a' or 'A' and are at least 3 characters long
'_____'	are exactly 5 characters long

For more information about using LIKE in a WHERE clause, see Operators Used in the WHERE Clause.

Negation

NOT can be used in an expression to reverse the boolean value of a logical expression.

For example:

```
NOT lastname='Brown'
```

You can also precede an operator with NOT to negate the operator. For example:

```
num NOT BETWEEN 1 and 100
num NOT IN (1, 100)
```

Compound Expressions

Compound expressions can be created by joining multiple individual expressions by using the AND or OR operators.

These operators have the following behavior:

- **AND**: The overall expression is true if and only if every individual expression is true.

- **OR**: The overall expression is true if at least one of the individual expressions is true.

Order of Operations

Operator precedence is the order in which an operator is executed.

In SQL the operator precedence is as follows:

1. Parentheses
2. Multiplication or division
3. Addition or subtraction
4. NOT
5. AND
6. OR

Parentheses can be used to override these rules of precedence as shown in the following example:

```
A + B * C
```

Performs `B * C` and then adds `A` to the result.

```
(A + B) * C
```

Performs `A + B` first and then multiply the result by `C`.

CASE Expressions

A CASE expression is used like an `if-then-else` construct to conditionally return a value. There are two forms of syntax for CASE expression, the simple form and the searched form. The simple form is used to test a single operand for equality against multiple expressions. The searched form of a CASE expression is more flexible and allows for testing multiple conditions.

For more information about using CASE expressions in the select list of SELECT statement, see The Select List.

Simple CASE Expression

The simple CASE expression implies equality (=) is used for comparisons. One <*common_operand*> is tested against multiple values. It is frequently used to transform one set of values to another longer form.

Syntax:

```
CASE <common_operand>
    WHEN <expression> THEN <result>
   [WHEN <expression> THEN <result>
    . . .]
        [ELSE <result>]
END
```

For example:

```
SELECT lastname,
    CASE gender
        WHEN ‘M’ THEN ‘Male’
        WHEN ‘m’ THEN ‘Male’
        WHEN ‘0’ THEN ‘Male’
        WHEN ‘F’ THEN ‘Female’
        WHEN ‘f’ THEN ‘Female’
        WHEN ‘1’ THEN ‘Female’
        ELSE ‘Unknown’
    END
FROM employees
```

Equality expressions cannot be used to test for NULL so you cannot use a simple CASE expression to test for NULL. You must use a searched CASE expression with IS NULL or IS NOT NULL when testing for NULL.

Searched CASE Expression

The searched CASE expression is good to use when you want to work with a greater range of tests. Any boolean expression qualifies as a WHEN <*expression*>.

Syntax:

```
CASE
    WHEN <condition> THEN <result>
   [WHEN <condition> THEN <result>
    . . .]
        [ELSE <result>]
END
```

The result of the first `true` boolean expression is returned. For example, the following

SELECT statement uses a searched CASE expression to prevent division by zero:

```
SELECT ProductID, Name, ProductNumber, Cost, ListPrice,
CASE
    WHEN ListPrice = 0 THEN NULL
    ELSE Cost / ListPrice
END AS CostOfGoodSold
FROM Product
```

The searched CASE expression can also be used to test a result column for NULL whereas a simple CASE expression cannot. The following simple CASE expression is invalid:

```
SELECT
CASE middle_name
    WHEN NULL THEN '<NULL>'
    ELSE
        middle_name
END
```

In SQL you cannot use equality to test if a column is NULL (empty). Instead you must use 'IS NULL' as follows:

```
SELECT
CASE
    WHEN middle_name IS NULL THEN '<NULL>'
    ELSE
        middle_name
END
```

CASE Comparisons

When you use a simple CASE expression or a searched CASE expression, the following behavior of comparisons is true:

- WHEN clauses are evaluated in the order they are defined.
- When an ELSE clause is provided and none of the previous WHEN clause evaluations match, the *ELSE result* is returned.
- If an ELSE clause is omitted and none of the previous WHEN clause evaluations match, NULL (SQL NULL) is returned.

SQL Functions

ActiveSpaces provides support for many functions that can be used in SQL statements, such as the aggregate functions, date and time functions, and string functions.

For information about using a function in a particular type of SQL statement, see The SQL SELECT Statement. If it does not include information on using a function in the section, then its use is not supported with that statement type. Aggregate functions are only used in the select list of SELECT statements. For more information about aggregate functions, see Aggregate Functions.

Aggregate Functions

Aggregate functions are used in the select list of SELECT statements and use the values of multiple rows to calculate a single value.

ActiveSpaces supports the following aggregate functions:

- COUNT
- SUM
- MIN
- MAX
- AVG
- GROUP_CONCAT

For more information about aggregate functions, see the section on "Aggregate Functions" in Functions Used in The Select List.

Performance of Aggregate Functions

When using an aggregate function as a value expression, the performance of the aggregation depends on how many rows are used in the aggregation. A query that includes aggregation, but does not contain a WHERE clause performs aggregation using all the rows from the table but must be avoided, if possible.

For the best aggregation performance, the column used as an argument to your aggregation function must have an index defined, which uses that column, and your query must include a WHERE clause that causes that index to be selected for the query. For example, if you have a secondary index named `lastname_idx` defined for the lastname column, the following queries use `lastname_idx` to reduce the number of rows in the query

on which aggregation is performed:

```
SELECT COUNT(lastname) FROM mytable WHERE lastname LIKE 'B%'
SELECT lastname, COUNT(lastname) FROM mytable WHERE lastname LIKE 'B%'
GROUP BY lastname
```

For more information about query performance see Tips on Constructing an Efficient WHERE Clause.

Timeouts

On large data sets, calculating the aggregate results can take a long time because large numbers of rows must be processed before the computation can be completed. In such cases, when creating or executing the statement, it is advisable to set `TIBDG_STATEMENT_PROPERTY_DOUBLE_FETCH_TIMEOUT` to a value that prevents timeouts. For example, if the client request timeout is `5` seconds, a more complex query can set the fetch timeout to a value larger than `5` seconds to prevent a timeout.

For more information about Statement properties, see Properties.

Floating Point Calculations in Aggregate Functions

By default, floating point calculations are inexact and the order in which they are carried out can subtly affect the results. When executing queries involving aggregate functions, ActiveSpaces processes the data from the different copysets in parallel leading to slight variations in the results of floating point calculations. Even if the variations might be very small, they can be amplified by rounding the decimal value. For example, the average of a column can be calculated as 80.2849999999999966 one time and 80.2850000000000108 another time. This variation is clearly very small (1.4e-14), but if the results are rounded to 2 decimal places, that is 80.28 and 80.29, the variation appears greater than that.

Proxy Binding

Applications that run large queries that use aggregation must work with their administrator to determine the best approach for binding to proxies in the system. Your administrator can recommend a proxy binding strategy that keeps computation intensive queries from interfering with queries that have to run more quickly by either isolating or distributing your queries across the proxies available to you.

Limitations of the Aggregate Functions

The current support for aggregate functions requires that the argument to each aggregate function must be a column name. For example:

```
AVG(col1)
```

The aggregate function argument cannot be an expression or function. For example, the following function arguments are invalid uses of aggregate functions:

```
AVG(col1 + col2)
MIN(trim(col1))
```

For SQL keywords that cannot be used with aggregation functions, see Unsupported SQL Features.

Date and Time Functions

ActiveSpaces provides support for several date and time functions. The date and time functions return either a string or a double.

The following is the list of date and time functions supported.

Date and Time Function	Returns
`date(timevalue[, modifier[, modifier, …]])`	The date as a string in the format YYYY-MM-DD
`time(timevalue[, modifier[, modifier, …]])`	The time as a string in the format HH:MM:SS
`datetime(timevalue[, modifier[, modifier, …]])`	The date and time as a string in the format YYYY-MM-DD HH:MM:SS
`julianday(timevalue[, modifier[, modifier, …]])`	The number of days since noon in Greenwich on November 24, 4714 B.C as a double.

The date and time functions can act on the values of columns that have been defined as one of the following data types: `string`, `double`, `long`, and `datetime`. The following table shows how each of the data types represent date and time:

Data Type	Representation of Date and Time
`string`	ISO_8601 strings. For example, "YYYY-MM-DD HH:MM:SS.SSS"

Data Type	Representation of Date and Time
`double`	Julian day numbers, the number of days since noon in Greenwich on November 24, 4714 B.C. according to the proleptic Gregorian calendar
`long`	(default) as Julian day numbers. See type double in the earlier row.
`long`	Unix Time, the number of seconds since 1970-01-01 00:00:00 UTC
`datetime`	two 64-bit integers; one for the number of seconds since January 1, 1970 (Unix epoch), and one for the number of nanoseconds after the time that the `sec` component denotes.

By default, a column of type long is treated as a Julian day number. A column of type long can also be used to hold Unix time values (e.g. the number of seconds since 1970-01-01 00:00:00 UTC). When using a date and time function on a column of type long that holds Unix time values, the 'unixepoch' modifier must follow the column name in the parameters passed to the function otherwise the column value is interpreted as a Julian day. See Modifiers.

The following are examples of using the date 2016-12-24 00:00:00 as values in the different column types:

- `string - "2016-12-24T00:00:00"`. For details, see timevalue Format.
- `double - 2457746.50000`
- `long - 1482566400 (Unix time)`
- `datetime - [1482566400, 0]`

timevalue Format

The `timevalue` parameter represents a date/time in a format understood by the date and time functions.

`timevalue` can be in any of the following formats:

1. YYYY-MM-DD
2. YYYY-MM-DD HH:MM
3. YYYY-MM-DD HH:MM:SS
4. YYYY-MM-DD HH:MM:SS.SSS

5. YYYY-MM-DD**T**HH:MM
6. YYYY-MM-DD**T**HH:MM:SS
7. YYYY-MM-DD**T**HH:MM:SS.SSS
8. HH:MM
9. HH:MM:SS
10. HH:MM:SS.SSS
11. D[D..][.D..]
12. now

When `timevalue` is the name of a column in a table, the value of the column is substituted for the `timevalue` parameter. As per ISO-8601, a date and time can be combined using the literal character `T`. The `T` can also be omitted by mutual agreement. Both ways of combining dates and times into a single string are supported.

Formats 2 through 10 can optionally be followed by a timezone indicator. The timezone indicator can have the following formats:

- [+-]HH:MM
- Z

Z represents UTC time and is the timezone used to store dates and times in the data grid. If HH:MM is non-zero, it is subtracted from the date and time and is intended to be used to convert to UTC time.

The fractional seconds value SS.SSS can have one or more digits following the decimal point but only the first three digits are considered. Therefore, support for full nanosecond precision of `tibDateTime` values stored into ActiveSpaces `datetime` columns must not be expected for queries.

Format 11 is the Julian day number expressed as a long or floating point number. This format can accept any number of digits as required to represent the Julian day number. Format 11 can also be a `long`, which represents Unix time. By default `long` values are interpreted as Julian days. The `unixepoch` modifier must follow the column name in the parameters passed to the date and time functions for the value of the column to be interpreted as Unix time. See the section Modifiers.

now is converted into the current date and time.

> **Note:** The SQL variables `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` can be used instead of specifying `date('now')`, `time('now')`, and `datetime('now')`, respectively.

Modifiers

Each date and time function accepts zero or more modifier parameters that can be used to alter the date or time returned by the function. Modifiers are applied from left to right in the order specified.

The following modifiers can be used to add to the date/time:

- NNN day[s]
- NNN hour[s]
- NNN minute[s]
- NNN.NNNN second[s]
- NNN month[s]
- NNN year[s]

The following modifiers shift the date or time backward:

- start of month
- start of year
- start of day

The weekday modifier can be used to shift the date forward to the next date when the weekday number is N. Sunday starts at 0, Monday is 1, and so on.

- weekday N

The following modifiers can be used to convert the `timevalue` immediately preceding it to something else. For each of these modifiers, the results are undefined if the `timevalue` preceding it is not of the proper type.

- unixepoch
- localtime
- utc

The unixepoch modifier causes the `timevalue` preceding it to be interpreted as Unix Time

(the number of seconds since January 1, 1970). The localtime and utc modifiers can be used to convert the `timevalue` immediately preceding it from UTC time to localtime or localtime to UTC time respectively.

Result Column Examples

The date and time functions supported by ActiveSpaces can be used in the list of result columns for a SELECT statement. The date and time functions cannot be used with table iterators.

Given a table with the following columns and values for 2016-12-24 00:00:00:

Columns	Data Type	Values
`dtm`	`datetime`	[1482566400, 0]
`dtmstr`	`string`	“2016-12-24T00:00:00”
`dtmlong`	`long`	1482566400
`julianday`	`double`	2457746.50000

The following are examples of SELECT statements that use the date and time functions in the select list of the SELECT statement:

```
SELECT key, date(dtm) FROM mytable WHERE key<=10
SELECT key, datetime(dtm) FROM mytable WHERE key<=10
SELECT key, time(dtm) FROM mytable WHERE key<=10
SELECT key, julianday(dtm) FROM mytable WHERE key<=10
SELECT key, datetime(now), date(dtm) FROM mytable WHERE key <=10
```

SQL String Functions

ActiveSpaces includes the following SQL String functions:

Function	Description
|| (concat operator)	Concatenates two strings and returns a single string. The ||

Function	Description
	operator joins together the two strings of its operands.
char(X1,X2,...,XN)	Returns a string composed of the characters represented by the Unicode code points specified by X1, X2, and so on.
instr(X,Y)	Finds the first occurrence of string Y in string X. If a match is found, the function returns a Long value, which is the starting position of string Y in string X. Otherwise, it returns 0. If either X or Y is NULL, then the result is NULL, which in SQL means no value is returned. Remember that in SQL, the first character is considered position 1 and not 0. For example: `SELECT instr(FIRST_NAME, 'jo') FROM t1` If the `FIRST_NAME` is 'john', the function returns the starting position of the string "jo " in the string "john". In this case, the function returns 1.
ltrim(X)	Returns a string formed after removing space characters, if any, from the left of string X.
ltrim(X.Y)	Returns a string formed by removing the characters that appear in Y from the beginning of X. For example, `SELECT ltrim(FIRST_NAME, 'j') FROM t1` In this case the function returns "ohn" since it removes the 'j' character from the left of the string.
rtrim(X)	Returns a string formed after removing space characters, if any, from the right of string X.
rtrim(X.Y)	Returns a string formed by removing the characters that appear in Y from the end of X.

Function	Description
	`SELECT rtrim(FIRST_NAME, 'n') FROM t1` In this case the function returns "joh" if the value in the column was "john".
trim(X)	Returns a string formed after removing space characters, if any, from both sides of string X.
trim(X,Y)	Returns a string formed by removing the characters that appear in Y from the beginning and the end of X. `SELECT trim(FIRST_NAME, 'n') FROM t1` In this case the function returns "atha" if the value in the column was "nathan".
lower(X)	Returns a string formed after converting the characters in string X to lower case.
upper(X)	Returns a string formed after converting the characters in string X to upper case.
length(X)	Returns a long that is the length of the string X. If X is NULL, the function returns NULL.
substr(X,Y)	Returns all characters through the end of the string X starting from position Y. The left-most character of X is at position 1. If Y is negative then the first character of the substring is found by counting from the right rather than the left. Characters indices refer to actual UTF-8 characters. For example: `substr('Hello World', 7)` Returns "World".

Function	Description
substr(X,Y,Z)	Returns a substring of input string X that begins at the position Y character and is Z characters long. The left-most character of X is at position 1. If Z is negative then the abs (Z) characters preceding the position Y are returned. For example: `substr('Hello World', 2, 4)` Returns 'ello'.
unicode(X)	Returns the numeric Unicode code point corresponding to the first character of the string X. If X is not a string, the result is undefined.

The ActiveSpaces JDBC Driver

Developers can use the ActiveSpaces JDBC driver to connect to an ActiveSpaces data grid. The ActiveSpaces JDBC driver implements the Java Database Connectivity (JDBC) API. The driver provides ActiveSpaces data grid connectivity for Java applications and third-party tools by using JDBC.

The ActiveSpaces JDBC driver is a Type 2 driver and makes native library calls for communicating with the data grid. The Java sample, `TIBCO_HOME/as/<version>/samples/src/java/com/tibco/datagrid/samples/ASandJDBCClient.java`, demonstrates the interaction between the ActiveSpaces JDBC driver and the data grid. A sample client application is provided that demonstrates using the JDBC driver to interact with an ActiveSpaces data grid in the Java samples shipped with ActiveSpaces.

The ActiveSpaces JDBC driver is not a fully compliant implementation of JDBC.

For more information about JDBC compliance, see JDBC Compliance.

For more information about the JDBC API, see *TIBCO ActiveSpaces® Java API Reference*.

Connecting to the Data Grid by Using ActiveSpacesJDBC Driver

Java applications can connect to the ActiveSpaces data grid by using the ActiveSpaces JDBC Driver.

Before you begin

Complete the steps in Setting up the Environment.

Procedure

1. Register the ActiveSpaces JDBC driver with the JDBC Driver Manager. For more information, see Registering the ActiveSpaces JDBC Driver with the Driver Manager.
2. Connect with an ActiveSpaces data grid by using the appropriate JDBC URL. For more information, see Creating the ActiveSpaces JDBC Connection.

Setting up the Environment

The ActiveSpaces JDBC driver is bundled with the ActiveSpaces Java client API and included in `tibdg.jar`. To use the ActiveSpaces JDBC driver from a client application, follow the same steps for setting up your application that you used for the Java API. For information about setting up your environment to build and run the ActiveSpaces samples, refer to the *TIBCO_HOME*`/as/<version>/samples/src/Java/README.md` file.

You can use the same environment for developing Java client applications. The ActiveSpaces JDBC driver is a Type 2 driver and uses native code for communicating with a data grid. When using the ActiveSpaces JDBC driver with client JDBC tools, you must configure the location of the ActiveSpaces native libraries.

On Linux/Mac, you can start the tool by either using the OS environment variable to specify the search path for native libraries (ex: LD_LIBRARY_PATH pointing to the AS lib directory) or by specifying that path with the following Java command-line options:

```
-Djava.library.path=${TIBDG_ROOT}/lib
```

On Windows, the `java.library.path` is not sufficient to find the dependent native libraries. You must include the paths to all native libraries (both AS and FTL bin directories) in the PATH environment variable and can avoid using the `java.library.path`.

After setting up the environment, Registering the ActiveSpaces JDBC Driver with the Driver Manager.

Registering the ActiveSpaces JDBC Driver with the Driver Manager

Use the class name, `com.tibco.datagrid.jdbc.DriverImpl` when registering the ActiveSpaces JDBC driver or configuring other software to use the ActiveSpaces JDBC driver.

Before you begin

Complete the steps listed in Setting up the Environment.

Procedure

1. Use the following code snippet to register the ActiveSpaces JDBC driver with the

JDBC DriverManager:

```
 // Register the ActiveSpaces JDBC Driver with the JDBC
DriverManager
            try
            {
                Class.forName
("com.tibco.datagrid.jdbc.DriverImpl");
            }
            catch (ClassNotFoundException ex)
            {
                // handle exception
            }
```

What to do next

After registering the driver, the next step is Creating the ActiveSpaces JDBC Connection.

Creating the ActiveSpaces JDBC Connection

After registering the ActiveSpaces JDBC driver, the next step is to make the ActiveSpaces JDBC connection.
The DriverManager interface supports the following methods for creating a connection:

```
Connection getConnection(String url);
Connection getConnection(String url, Properties info);
Connection getConnection(String url, String user, String password);
```

The `url` parameter is the JDBC URL to connect to a database, or in our case a data grid. Ensure that the `url` adheres to the format specified in JDBC URL Format. The `info` parameter is optional and can contain a set of Java properties that are required for connecting to the data grid. To understand these properties, see ActiveSpaces Connection Properties .

Note: Properties passed to the `connect()` method override the properties specified in the JDBC URL. The names of the properties follows the same rules for property names in the JDBC URL.

Before you begin

First complete the steps listed in Registering the ActiveSpaces JDBC Driver with the Driver Manager.

Procedure

1. After registering the driver, create a JDBC connection to the data grid. To connect to the data grid, you can use any of the `getConnection()` methods. The following code snippet uses `Connection getConnection(String url)` :

```
// create a JDBC connection to the data grid
        String jdbcURL = "jdbc:tibco:tibdg:_default;realmurl=
        http://localhost:8080";
        java.sql.Connection jdbcConnection =
        java.sql.DriverManager.getConnection(jdbcURL);
```

An Example of Registering and Connecting to the Data Grid

This example shows the code snippet to be inserted to your Java code to register and connect to the data grid. The following example uses the `Connection getConnection (String url, Properties info)` method to create a connection.

```
public java.sql.Connection getConnection (Properties props) throws
SQLException
        {
            // Register the ActiveSpaces JDBC Driver with the JDBC
             DriverManager
            try
            {
                Class.forName("com.tibco.datagrid.jdbc.DriverImpl");
            }
            catch (ClassNotFoundException ex)
            {
                // handle exception
            }

            // Establish a JDBC connection to the data grid
            String jdbcURL = "jdbc:tibco:tibdg:_default";
            Properties props = new Properties();
            props.setProperty(
            com.tibco.datagrid.Connection.TIBDG_CONNECTION_PROPERTY_
STRING_REALMURL,
            "http://localhost:8080");

            return java.sql.DriverManager.getConnection(jdbcURL, props)
        }
```

JDBC URL Format

You specify a JDBC URL when you are establishing a JDBC connection to an ActiveSpaces data grid.

The standard syntax for JDBC URLs is:

```
jdbc:<subprotocol>:<subname>
```

The variables in this syntax are as follows:

- `subprotocol` is the name of the driver or the name of a database connectivity mechanism.
- `subname` is a way to identify the data source.

The ActiveSpaces JDBC URL has been extended from the standard JDBC URL syntax to provide you the ability to specify the settings for connecting to an ActiveSpaces data grid. The ActiveSpaces JDBC URL format is as follows:

```
jdbc:tibco:tibdg[:<data-source-name>][;<propertyName>=<propertyValue>]*
```

The *<data-source-name>* is optional and specifies the data grid name. The data grid name can also be specified as a property as in the following example URL:

```
jdbc:tibco:tibdg;gridname=mygrid
```

If the data grid name is not specified as a property or as *<data-source-name>*, then `_default` is used, which is the default data grid name. For example:

```
jdbc:tibco:tibdg
```

ActiveSpaces Connection Properties

The ActiveSpaces API consists of several properties that can be set when connecting to an ActiveSpaces data grid. These properties control how to connect to the data grid and some of the behaviors of the data grid.

The following partial list describes the connection properties that can be specified. The full list is in the Javadoc for `com.tibco.datagrid.Connection`.

Property	Description	Default Value
`com.tibco.tibdg.gridname`	Name of the data grid	`_default`
`com.tibco.tibdg.realmurl`	Realm Service URL. The Realm Service URL use the pipe character (|) as a separator to provide a list of URLs.	`http://localhost:8080`
`com.tibco.tibdg.connectwaittime`	The amount of time to wait for connection to be established.	`0.1 seconds`
`com.tibco.tibdg.timeout`	The amount of time to wait for responses from the data grid.	`5.0 seconds`

When specifying properties as part of the JDBC URL, the full property name or the property name without the `com.tibco.tibdg` prefix can be used. For example:

```
jdbc:tibco:tibdg:_default;com.tibco.tibdg.realmurl=http://localhost:8080
```

Using the ActiveSpaces JDBC Driver With Third Party Tools

Date and time values stored in ActiveSpaces are stored in UTC time. By default, third party JDBC clients use the default timezone of the JVM where the client was started when retrieving date and time values. To see dates and times in a particular time zone when they are retrieved from the data grid, you can use one of the JDBC methods that helps you specify a time zone. Another alternative is to configure your JDBC client to display date and time information in UTC time. For example, when using the SQuirreL JDBC client you can modify the script file used to start SQuirreL and specify the following additional command-line option where the SQuirreL executable is invoked:

```
-Duser.timezone=UTC
```

JDBC Implementation Notes

This section focuses on some important aspects to consider when using the ActiveSpaces JDBC driver.

JDBC Data Types

ActiveSpaces uses a small set of data types for storing data in a data grid. When using the ActiveSpaces JDBC driver, it is important to know how the JDBC data types map to the ActiveSpaces data types.

The following table lists the JDBC data types and how they map to the ActiveSpaces data types:

JDBC Data Type	ActiveSpaces Data Type
CHAR, VARCHAR, LONGVARCHAR	string
BIT, BOOLEAN, TINYINT, SMALLINT,INTEGER, BIGINT	long
REAL, FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY,BLOB	opaque
DATE, TIME, TIMESTAMP	TibDateTime

Any JDBC data types not listed above are not supported by the ActiveSpaces JDBC driver.

A table defined in ActiveSpaces has the data types of its columns mapped to the following JDBC data types when using the JDBC driver:

ActiveSpaces Data Type	JDBC Data Type
string	VARCHAR
long	BIGINT

ActiveSpaces Data Type	JDBC Data Type
double	DOUBLE
opaque	VARBINARY
TibDateTime	TIMESTAMP

DatabaseMetaData Pattern Parameters

Methods of the `DatabaseMetaData` interface that take arguments that are string patterns, it must be noted that the pattern `"%"` are matched only when used by itself.

Here is an example:

Code Snippet	Returns
getTables(null, null, "%", null)	a `ResultSet` containing all tables in the data grid
getTables(null, null, "table%", null)	an empty `ResultSet` as `"%"` is not used by itself for the pattern argument

ResultSetMetaData and Function Return Values

ActiveSpaces determines the data types of the return values of functions used in queries at runtime. When the `ResultSet` for a query contains the return value of a function, VARCHAR is always reported as the data type for the value in the `ResultSetMetaData`.

To retrieve a function return value from the `ResultSet` as its actual data type, you must wait until the query is executed. After the query has been executed, the custom method `ResultSetImplDG.getColumnType()` can be called, which returns the actual result value type.

For example, suppose you have the following query:

```
SELECT COUNT(key) FROM mytable
```

The `ResultSetMetaData` reports that the data type for column 1 is VARCHAR. To retrieve the actual data type for column 1 of the query result do the following:

1. Unwrap the `ResultSet` object retrieved when the query is executed. For example,

```
ResultSetImplDG rsImpl = resultSet.unwrap
(com.tibco.datagrid.jdbc.ResultSetImplDG.class);
```

2. Retrieve the data type of column 1 in the `ResultSet`. For example,

```
int columnType = rsimpl.getColumnType(1);
```

   The `columnType` must be set to a value from java.sql.Types to indicate the type of the value in the column. The `java.sql.Types.NULL` value is returned for columns in the row, which do not contain a value.

JDBC Compliance

The ActiveSpaces JDBC driver is not a fully compliant implementation of JDBC.

SQL command support is provided by ActiveSpaces and the driver does not implement any additional support. The following SQL commands are supported:

- CREATE TABLE
- CREATE INDEX
- DROP TABLE
- DROP INDEX
- SELECT
- INSERT
- INSERT OR REPLACE
- DELETE

The following are the known limitations of the driver:

- Entry Level SQL92 is not fully supported.
- Transactions are not supported.
- Batch updates are not supported.
- `javax.sql.DataSource` is not implemented.

The following JDBC driver features are not supported:

- Connection Pooling
- Schemas
- Foreign Keys
- Procedures
- Functions
- Callable Statements
- Savepoints
- Parameter Metadata
- Named Parameters
- Transactions
- Updatable ResultSets
- Multiple ResultSets From a Single Execute Call
- Joins
- Subqueries

Sizing Guide

Usually, the total data set is partitioned horizontally into copysets where each copyset holds a fraction of the data. Since a copyset in production typically includes more than one node for redundancy (where each node is an exact replica of the data in that copyset), let us start with a simplifying assumption that the data resides on a single node per copyset.

The size of a copyset is determined by the following factors:

- The number of rows
- The size of a row in bytes (The size of a row is determined by number of columns, the column data types, and the actual values placed in each column)
- Indexes

ActiveSpaces provides an Excel spreadsheet that can be used to calculate the size of your data grid. Download the *ActiveSpaces Sizing Guide* from the TIBCO ActiveSpaces landing page. Please review the spreadsheet for information about how the number of bytes in the example used in this guide were determined.

Example of a Sizing Calculation

Consider a scenario where the purchasing details of a customer are stored in the purchase table. There are around five million rows in this table with the following schema:

Name of the field	Data Type	An estimation of the disk space consumed (in bytes)
customer_id (Primary Index)	Long	8
purchase_id	Long	8
customer_first_name	String	10
customer_last_name	String	10

Name of the field	Data Type	An estimation of the disk space consumed (in bytes)
customer_post_code	Long	8
payload	String	10K

Size of Rows without Secondary Indexes

Size of Row (in bytes) = 8 + 8 + 10 + 10 + 8 + 10K = 10,044

Estimated Internal Overhead + Primary Index (Long) Overhead per Row = (32 + 27) = 59 bytes

Size of Row Including Overhead = 10,103

Size of All Rows with No Secondary Indexes = 5M x 10,103 = 50.5GB

Size of Rows with Secondary Indexes

Index Overhead per Row = 45 bytes (might vary depending on actual values being indexed)

purchase_id_idx = 5M x (45 + 8) = 0.27GB

customer_full_name_idx = 5M x (45 + 10 + 10) = 0.33GB

Size of Secondary Indexes = 0.6GB Total Size In Bytes

(All Rows + Secondary Indexes) = 51.1GB

Additional Factors That Affect Sizing

The purchase table with five million rows would be estimated to occupy 51.1GB on disk. You must provision additional disk space to account for compactions and other internal activity. Depending on the configuration options, you might need 10% to 100% additional space on disk. The additional disk space needed depends on whether you opt for better write performance by not using compaction and providing more disk space reducing the amount of disk space required by using compaction. The different compaction levels can be used to reduce the actual disk space requirements of a typical 10K XML string when reducing performance. For better performance, apply the 100% Free Space Factor from the Excel spreadsheet, which results in an estimated 51.1GB of additional free disk space needed.

Determining the Allocation of Physical Computers to the Nodes

When we map the disk space needed to node processes running on the hardware, a general guideline would be to start with two copysets. Each copyset would own half the data (approximately 26GB) and each copyset would be configured with at least two nodes for redundancy (where each node would have to hold the full amount of data owned by that copyset). As a result, on each copyset, you would have two copies of the data for redundancy.

Here is how this would map to four node processes (where each node process would be run on a separate computer):

Copyset1

Node = 26GB

Node = 26GB

Copyset2

Node = 26GB

Node = 26GB

After adding a multiplier (Free Space Factor times node data size) = approximately 52GB.

Apply the multiplier to ensure that there is enough free disk space. A general guideline is to double all of these numbers to have approximately 52GB on each node.

However, there is no strict guideline for the amount of RAM needed based on the amount of data. If there is available RAM, it would be used for caching. In this case, given that nearly all the data must be able to fit in RAM on each node, you can opt for 32GB or 64GB of RAM on each of the nodes.

Similarly, you can opt for an SSD that can account for the data and free space such as 256GB SSD on each computer.

Remember that every node must be capable of holding the amount of data held by the copyset. After determining the size of a node, the next step is to decide how many nodes you want in a copyset. The number of nodes depends on how many replicas of data you want to maintain and how many copysets you want to have in the data grid.

After you have an estimate about the number of nodes and replicas of data, you can use the following formula to determine the number of nodes in a data grid:

```
Total number of nodes in the data grid = Number of copysets *
number of replicas in a copyset
```

Note: ActiveSpaces 4.0 and later does not require each node to have as much RAM as the full amount of data.
However, you can provision them that way for optimal read performance depending on read access patterns by the application.

Comparison Matrix

This matrix lists the major differences in the terminology used in ActiveSpaces 2.x and 3.x and later. This is not an exhaustive matrix and only highlights the salient differences.

ActiveSpaces 2.x or earlier	ActiveSpaces 3.0 and later
Metaspace, cluster	Data grid
Space	Table
Tuple	Row
Attribute	Column
Key	Primary Index
Seeder	Node
Leech	N/A
Remote clients	ActiveSpaces clients

Functionality in ActiveSpaces 3.0 and Later That Is Different from Earlier Versions

No Automatic Redistribution

When a new node is added to the data grid, redistribution of rows does not happen automatically and thus prevents the system from slowing down. The Administrator is in total control of manual redistribution.

Data Ownership

In ActiveSpaces 3.0 or later, data is organized into logical copysets and the nodes in a copyset are all identical replicas. In ActiveSpaces 2.x, a consistent hashing algorithm is used to determine which seeder in the cluster owns a given tuple.

Supported Primitive Operations

ActiveSpaces 3.0 or later supports the following primitive operations: GET, PUT, and DELETE.

ActiveSpaces 4.5 or later supports the following primitive operations: GET, PUT, UPDATE, and DELETE.

No Shared All Mode of Persistence

Nodes in ActiveSpaces 3.0 or later store their data locally to the disk. Storing locally is equivalent to Shared Nothing persistence with ActiveSpaces 2.x.

Error Codes

The following table lists the error codes used in ActiveSpaces:

ActiveSpaces relies on TIBCO FTL internally. TIBCO FTL generates error codes ranging between `-1` and `5000`. For information about TIBCO FTL error codes, see TIBCO FTL Error Codes.

Error Code	Error Key	Cause	Solution
-1	TIB_NULL_ EXCEPTION	The program has supplied a NULL value instead of an exception object.	
0	TIB_OK	The TIBCO FTL call completed successfully.	
1	TIB_INVALID_ ARG	An invalid value has been embedded as an argument.	
2	TIB_NO_MEMORY	An application could not allocate sufficient memory to process an operation.	
4	TIB_TIMEOUT	Internal timeout elapsed.	In many cases the library attempts to retry most operations but stops once the client's timeout period is reached. An application may choose to retry the function a certain number of times before taking some other course of action.
5	TIB_NOT_ INITIALIZED	The program has not yet called `tib_Open` to start	In most cases the application must retry an operation,

Error Code	Error Key	Cause	Solution
		TIBCO FTL or grid is offline.	maybe with a brief delay, some number of times till the grid is online again.
6	TIB_OS_ERROR	An operating system call has failed.	
7	TIB_INTR	An internal interrupt has occurred in a thread.	
8	TIB_NOT_ PERMITTED	An invalid operation has occurred on a valid object. You don't have a permission to perform the operation.	To perform the operation, get a permission either directly or through a group of which you are part of.
9	TIB_NOT_FOUND	A client has requested a value that is not in the database.	
10	TIB_ILLEGAL_ STATE	The action has been inconsistent with the internal state.	
11	TIB_NOT_ SUPPORTED	The data type or feature is not supported.	
12	TIB_END_OF_ BUFFER	Reached the end of buffer when parsing an inbound message.	
13	TIB_VERSION_ MISMATCH	Incompatible versions of TIBCO FTL components.	
14	TIB_ALREADY_ EXISTS	There has been a conflict with an existing object, value, or definition.	

Error Code	Error Key	Cause	Solution
15	TIB_FILE_IO_ ERROR	I/O error occurred when accessing shared memory or when accessing the file system.	
16	TIB_INVALID_ VALUE	TIBCO FTL internal error has occurred due to one of the following malformed artifacts: message, packet, or realm definition.	
17	TIB_INVALID_ TYPE	There has been a mismatch between field and data type.	
18	TIB_INVALID_ CONFIG	The property values have been invalid or contradictory.	
19	TIB_INVALID_ FORMAT	The FTL client library has encountered a message with an invalid format.	
20	TIB_CLIENT_ SHUTDOWN	The program has attempted an operation on an unusable realm object.	
21	TIB_RESOURCE_ UNAVAILABLE	A resource required by the client program was unavailable.	
22	TIB_LIMIT_ REACHED	A resource could not accept data because it had reached a specified upper limit.	
23	TIB_FORMAT_	A format required by the	

Error Code	Error Key	Cause	Solution
	UNAVAILABLE	client program was unavailable.	
100	TIB_EXCEPTION	An unclassified exception occurred.	
101	TIB_UNKNOWN_SYSPROP	The client has detected a corrupted message during reassembly.	
5000	TIBDG_INVALID_BIN_EPOCH	Could have occurred after a successful redistribution when a stale operation was sent to the old node, which previously owned the row but no longer has the ownership. Retry the operation again so that it goes to the new owner.	
5002	TIBDG_INVALID_RESOURCE	This error code is returned when a specific resource is to be recreated. If a `Prepared Statement` exists before redistributing a grid, subsequent `ExecuteQuery()` calls might fail.	In this case, the `Prepared Statement` must be closed and recreated before running any more `ExecuteQuery()` commands.
5003	TIBDG_GRID_IN_MAINTENANCE	An invalid operation has been submitted in maintenance mode. This error code is returned when a client submits any operation that modifies the data on a disk while the grid is in maintenance mode.	The application must retry the operation until the grid is no longer in maintenance mode and the operation succeeds.

Error Code	Error Key	Cause	Solution
	TIB_INVALID_ARG	One of the arguments provided to the function is not valid.	Call the function again with same or different arguments.

SQL Error Codes

Error Code	Error Key	Cause	Solution
5100	TIBDG_SQL_SYNTAX_ERROR	An error is detected in the syntax of a SQL statement.	Correct the SQL statement before submitting it again.
5101	TIBDG_SQL_NOT_SUPPORTED	The string containing a SQL statement has included the syntax of an unsupported feature.	Correct the SQL statement before submitting it again.
5102	TIBDG_SQL_INVALID_VALUE	The string containing an SQL statement has included syntax that cannot be used in a particular scenario.	Correct the SQL statement before submitting it again.
5103	TIBDG_SQL_PARSER_ERROR	An unexpected error has occurred during the parsing of a SQL statement.	Correct the SQL statement before submitting it again.
5104	TIBDG_SQL_STMT_ERROR	An unexpected error has occurred during the processing of a SQL statement.	The statement should be resubmitted as the statement is most likely valid.
5105	TIBDG_SQL_SYSTEM_ERROR	An unexpected SQL system error occurred during the processing of a query.	The statement should be resubmitted as the statement is most likely valid.

Error Code	Error Key	Cause	Solution
5106	TIBDG_SQL_QUERY_ERROR	An unexpected error occurred during the processing of a query by a `tibdgnode`.	The statement should be resubmitted as the statement is most likely valid.
5107	TIBDG_SQL_DDL_CMD_ERROR	An unexpected error occurred during the processing of a table or index DDL command in the data grid.	The statement should be resubmitted as the statement is most likely valid.
5108	TIBDG_SQL_QUERY_CMD_ERROR	An unexpected error occurred during the processing of a query by the node.	Correct the SQL statement before submitting it again.
5109	TIBDG_SQL_DML_CMD_ERROR	An unexpected error occurred during the processing of an INSERT or UPDATE command in the data grid.	Correct the SQL statement before submitting it again.
5110	TIBDG_SQL_CMD_ERROR	A generic SQL command error occurred during the processing of a SQL command in the data grid.	Running the command again might result in it being successfully processed. Correct the SQL statement before submitting it again.
5111	TIBDG_SQL_NOT_PERMITTED	A SQL statement tried to perform an action that is not permitted by the data grid.	Running the command again is unlikely to be successful. Correct the SQL statement before submitting it again.
	TIBDG_SQL_BAD_BATCH	This error code will be returned when a batch of SQL commands could not be successfully processed due to one or more	When this happens, the entire batch of commands is rolled back.

Error Code	Error Key	Cause	Solution
		bad commands in the batch.	
	TIBDG_SQL_ RESOURCE_ UNAVAILABLE	when a SQL command could not be successfully processed due to a resource needed by the command being unavailable.	Running the command again might result in it being successfully processed.

TIBCO Documentation and Support Services

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit https://docs.tibco.com.

Product-Specific Documentation

The following documentation for TIBCO ActiveSpaces® is available on the TIBCO ActiveSpaces® Product Documentation page:

- TIBCO ActiveSpaces® *Release Notes*
- TIBCO ActiveSpaces® *Installation*
- TIBCO ActiveSpaces® *Concepts*
- TIBCO ActiveSpaces® *Administration*
- TIBCO ActiveSpaces® *API Reference*
- TIBCO ActiveSpaces® *Security Guidelines*
- TIBCO ActiveSpaces® *ActiveSpaces4-Sizing-Guide*

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit http://www.tibco.com/services/support.
- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support portal at https://support.tibco.com.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to https://support.tibco.com. If you do not have a user name, you can request one by clicking Register on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to https://community.tibco.com.

Legal and Third-Party Notices

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

This and other products of TIBCO Software Inc. may be covered by registered patents. Please refer to TIBCO's Virtual Patent Marking document (https://www.tibco.com/patents) for details.