TABLE OF CONTENTS

Contents:

Panther

JDB SQL Reference

Prolifics

Release 5.51

Document 0404
March 2017

Copyright

Send suggestions and comments regarding this document to:

Technical Publications Manager	http://prolifics.com
Prolifics, Inc.	support@prolifics.com
24025 Park Sorrento, Suite 405	(800) 458-3313
Calabasas, CA 91302	

TABLE OF
CONTENTS

Contents:

PREFACE

About This Document

JDB is a single-user relational database included with Panther. This guide provides information about relational databases in general, and about JDB specifically, including JISQL, the interactive SQL utility provided with Panther that lets you execute SQL.

For a description of relational databases, refer to Chapter 2, “Introduction to Databases.”

For information on building SQL statements, refer to Chapter 3, “Introduction to SQL.”

For information on JISQL, refer to Chapter 5, “Using JISQL.”

For complete information on SQL syntax in JDB, refer to Chapter 6, “SQL Reference.”

Documentation Website

The Panther documentation website includes manuals in HTML and PDF formats and the Java API documentation in Javadoc format. The website enables you to search the HTML files for both the manuals and the Java API.

Panther product documentation is available on the Prolifics corporate website at http://docs.prolifics.com/panther/.

How to Print the Document

You can print a copy of this document from a web browser, one file at a time, by using the File→Print option on your web browser.

A PDF version of this document is available from the Panther library page of the documentation website. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe website at https://get.adobe.com/reader/otherversions/.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously. Initial capitalization indicates a physical key.
italics	Indicates emphasis or book titles.
UPPERCASE TEXT	Indicates Panther logical keys. *Example*: XMIT

Convention	Item
boldface text	Indicates terms defined in the glossary.

`monospace text`	Indicates code samples, commands and their options, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <smdefs.h>`
	`chmod u+w *`
	`/usr/prolifics`
	`prolifics.ini`

`monospace italic text`	Identifies variables in code representing the information you supply. *Example*: `String` *`expr`*

`MONOSPACE UPPERCASE TEXT`	Indicates environment variables, logical operators, SQL keywords, mnemonics, or Panther constants.
	Examples:
	`CLASSPATH`
	`OR`

`{ }`	Indicates a set of choices in a syntax line. One of the items should be selected. The braces themselves should never be typed.
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `formlib [-v]` *`library-name`* `[`*`file-list`*`]...`

Convention	Item
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `formlib [-v] library-name [file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contact Us!

Your feedback on the Panther documentation is important to us. Send us e-mail at support@prolifics.com if you have questions or comments. In your e-mail message, please indicate that you are using the documentation for Panther 5.50.

If you have any questions about this version of Panther, or if you have problems installing and running Panther, contact Customer Support via:

- Email at support@prolifics.com
- Prolifics website at http://profapps.prolifics.com

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address and phone number
- Your company name and company address
- Your machine type
- The name and version of the product you are using

- A description of the problem and the content of pertinent error messages

CHAPTER

1 Introduction to JDB

JDB is a single-user Relational Database Management System (RDBMS). You can use it for building application prototypes; design your application and test data entry without requiring an external database. You can take full advantage of JPL procedures using standard SQL syntax as well as the transaction manager to access your JDB database information.

With JDB, you can:

- Create a database using JDB.
- Create menus and screens for an application using Panther.
- Enter data into your JDB database using Panther's database driver for JDB.
- Generate reports of information in your database.
- Test the entire application, including data entry. You can virtually do all you need to do with JDB that you would do with a standard database like Sybase or Oracle. You can also use the interactive SQL editor JISQL to gain direct access to the JDB data using standard SQL commands.

Using JDB

To use JDB, you build the database, connect to the database via the editor and import the database tables to a repository. Then you can begin building screens with the editor or screen wizard using the database-derived widgets in your repository.

All of the logic associated with the application resides either on your application screens, widgets, or service components–so when you connect to the actual database that your application will use, the application will work just as you prototyped it.

JDB Executables and Utilities

By default, your Panther executables are linked to JDB. For details on building an executable, refer to page 42-1 in the *Developer's Guide*.

The utilities you use to build and manipulate your JDB database are located in the specified directory of your Panther installation and are listed in Table 1.

Table 1-1 JDB-related utilities

Utility	Found in	Description
`isql`	`$SMBASE/jdb/bin`	Interactive SQL editor
`jdbroll`	`$SMBASE/jdb/bin`	Run log files
`jisql`	`$SMBASE/util`	Graphical interactive SQL editor
`mksql`	`$SMBASE/jdb/bin`	Translate JDB database into its related `CREATE TABLE` and `INSERT` statements
`tbldata`	`$SMBASE/jdb/bin`	Imports/exports database information

Under Windows, the graphical interactive SQL editor can be accessed via the JISQL icon (or under the Panther Start Menu option).

Unsupported Features

The following database features are not supported by JDB:

- Concurrency controls/locking
- Indexes
- Outer joins
- Stored procedures
- Triggers
- Views

See Also

Refer to page 5-1 for more information on JISQL. If you already have data that you want to enter into a JDB database, a utility is available to transfer data into a database table from an ASCII text file. Refer to page A-1 for more information on JDB utilities.

See Also

CHAPTER

2 Introduction to Databases

A database is a collection of information organized into different areas. Generally, a database covers information about a specific subject. For example, a company might have one database for personnel and another database for customer orders.

What sets a database system apart from other computer applications is that a structure exists which organizes the information. This structure allows each piece of information to be tagged. In some database systems, this structure is called the data model or database schema. Since there is a structure, the information stored in a database can be easily accessed for display to a screen or for printing in a report.

Structure of a Relational Database

To describe the structure of JDB, you need to have a basic understanding of relational databases as a whole.

Tables

JDB, like other relational database management systems, organizes its information into tables, which consist of a row and column arrangement of data values. Generally, a database table contains a subset of related information about the main subject.

For example, if inventory is the main subject for your database, you might have tables for products, orders, and suppliers.

The two-tier sample application provided with Panther uses a JDB database called `videobiz`. This database was designed for a video rental store, so it needs information about customers, video titles, and video rentals. Table 2-1 lists the database tables in the `videobiz` database.

Table 2-1 Videobiz database tables

Table Name	Description
`customers`	Address, phone number and membership information for each customer
`titles`	Title, director, length, rating and price category for each video.
`title_dscr`	Description of each video
`tapes`	Status of each copy of a video
`pricecats`	Listing of the various price categories
`actors`	Actors appearing in the videos
`roles`	Roles associated with each video
`rentals`	Video title, customer, and date information for each video rental
`codes`	Listing of the various codes used in the database
`users`	Login names for users of the database
`flag`	Yes/No flag used in the VideoBiz application

Columns

Each database table is divided into columns and rows. The columns are the various subcategories of the table, each containing a piece of the table information. The columns in the `titles` table, described in Table 2-2, have information about a video title, such as the director, the type of video, and the running time.

Table 2-2 Titles table

Column Name	Description	Status
`title_id`	Identification code for the video	Primary key
`name`	Name of the video	
`genre_code`	Code describing the video type	
`dir_last_name`	Director's last name	
`dir_first_name`	Director's first name	
`film_minutes`	Length of the video	
`rating_code`	Rating code	
`release_date`	Year the original film/video was released	
`pricecat`	Price category used when this video is rented	Foreign key

Entering Data

When you start to enter data into your database tables, you will not always have a value for every column. In those cases, the value of the column is said to be `NULL`. However, you must enter a value for your primary key columns and any column specified as `NOT NULL`. Those columns cannot have null values.

Null values are used when the column value is unknown or unavailable. A null value is not synonymous with an entry of zero or with a blank.

Rows

When you insert information into a database table, you can enter a value for each column. Each entry is called a row. In some database systems, the equivalent of a column is called a field and the equivalent of a row is called a record.

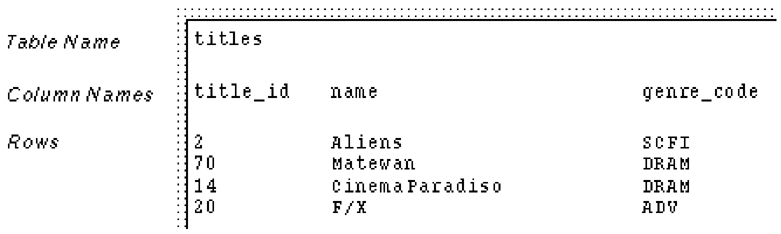


Table Name
Column Names
Rows

titles

title_id	name	genre_code
2	Aliens	SCFI
70	Matewan	DRAM
14	CinemaParadiso	DRAM
20	F/X	ADV

Figure 2-1 The columns and rows associated with the titles table.

Each row of data in a JDB database table can be up to 1K. For more information, refer to page 6-12.

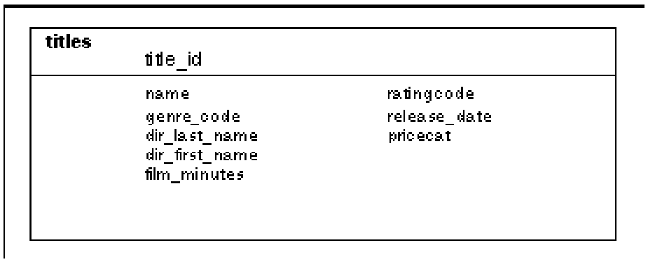
Primary Keys

Every table in a well-designed relational database should have a primary key. A primary key is the column, or set of columns, that uniquely identifies a row. Because the rows of a relational database are unordered, the primary key definition lets you select a particular row of information.

JDB does not enforce unique entries for the primary key columns. So, while you will not receive an error, you should define the primary key columns when you create your database tables because:

- The primary key definitions are assigned to table view properties when you import database tables to the repository.
- The transaction manager gives an error message if you attempt to insert a duplicate primary key or update the primary key column to a duplicate value.
- If you transfer your database schema to another database engine, the primary key columns will, no doubt, need to be defined for that engine.

JDB stores the primary key information. To define primary key information, choose which column or columns are the primary keys when you create your database tables. In the `titles` table, the primary key is the `title_id` column, therefore, a unique value should be applied for each row in that column.



In some database tables, a combination of two columns must make up the unique value. For example, the tapes table contains a `title_id` column, but a store can have several copies of a video title. Therefore, a combination of the `title_id` and `copy_num` columns must be used as the primary key of the tapes table. Every tape in the table is guaranteed to have a unique combination of data values in these two columns.

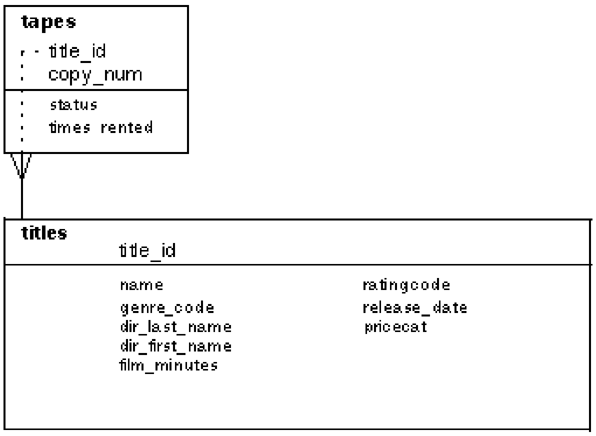
No two rows of a table with a primary key are duplicated. Table 2-3 lists the columns associated with the tapes table.

Table 2-3 Tapes table

Column Name	Description	Status
`title_id`	Title	Primary key, Foreign key
`copy_num`	Copy number for this tape.	Primary key
`status`	Code detailing whether the tape is available.	
`times_rented`	Number of times this copy has been rented.	

Foreign Keys

If a column in a database table matches the primary key in another table, the column is referred to as a foreign key. Any value entered into a foreign key column must match a value previously entered into the primary key column in the other table. For example, the `title_id` column is a foreign key in the `tapes` table. Therefore, any value entered into the `title_id` column of the `tapes` table should already exist as a value in the `titles` table.



Naming Database Tables and Columns

In JDB, names for tables and columns are one-word descriptions consisting of letters, numbers and underscores. Names can be up to 31 characters in length. Tables names must be unique within the database. Column names must be unique within the table.

When you create columns in a table, you tell the database whether the data in the column are character strings, dates, or numbers. For numbers, you specify what type can be entered–integer or float, for example. For character strings, specify the maximum length. The maximum allowable length for a character string column is 255.

Notes: Do not use JDB keywords as a name of a table or column; refer to page C-1 for a list of keywords.

Designing Your Database

The design process for building a database follows these general steps:

1. Choose the main subject.
   Decide the main focus of the database, for example, customer orders, inventory, or personnel. Generally, the database name indicates its main purpose. The two-tier sample application included with Panther is for a video rental store and is called videobiz.

2. Build your database model.
   Identify the database tables and their columns. Determine the main subsets of information; these correspond to the database tables. Then, decide which pieces of information are to be stored in each table; these pieces become the columns in each table.

3. Determine where common data values exist and eliminate duplicate data entry.
   Although a database table can contain all the information that logically relates to a subset of your database, this is not always the case. The database table should be designed to avoid duplicate data entry wherever possible. For example, in the sample application, information about video tapes is divided into two separate tables: titles and tapes. The titles table contains the information about the video title–its name, director, length, etc. The tapes table contains the information about each copy of the video–the copy number, status code, and the number of times this copy has been rented. By having two tables, you do not have to reenter the general video information, like the director and the length, for each copy of the video. You simply enter the `title_id`.
   If you split the information into two or more tables, determine which column will be found in all of the tables. In the sample application, the `title_id` column is found in several tables. This common data value in different tables provides the relationship needed to join tables together in order to combine the information.

In addition, you can also combine data from two tables into a single database table. For example, in the sample application, instead of having separate database tables, one for credit card codes and another for genre codes, the codes are combined into a single table of codes (called codes) used by the application. Entries into the columns `code_type`, code, and `dscr` describe and identify each specific code.

4. Define unique entries.
   For each table, determine which column or columns comprise the primary key, uniquely identifying each row. Since the titles table has more than one row with an entry of Henry V in the name column, the following statement affects both rows:

   ```
   DELETE FROM titles WHERE name = 'Henry V';
   ```

5. For the `titles` table, the `title_id` column is used to uniquely identify each row.

   To uniquely identify each copy of a video tape, the tapes table uses the `title_id` and `copy_num` columns to make the primary key. The following statement affects all rows with a value of 1345 in the `title_id` column:

   ```
   DELETE FROM tapes WHERE title_id = 1345;
   ```

   To change data about one copy of the tape, you need to list both the `title_id` and the `copy_num` columns in the `WHERE` clause.

   ```
   DELETE FROM tapes WHERE title_id = 1345 AND copy_num = 4;
   ```

6. Chart the tables and their relationships.

   Since information is stored in tables and the tables do not have any inherent relationship, it is possible to update a column in one table and not update the column in another table even if both columns correspond to the same value. To preserve the integrity of the database, it helps to chart the relationships between the tables. Then, if you update the information in one table, the chart illustrates if, and which, tables also need to be updated.

   Figure 2-2 is a diagram that describes the videobiz database tables and their relationships to one another. Each table is represented in a box. The table name and primary key columns are listed in the top of each box. The lines between the boxes illustrate the foreign key definitions.

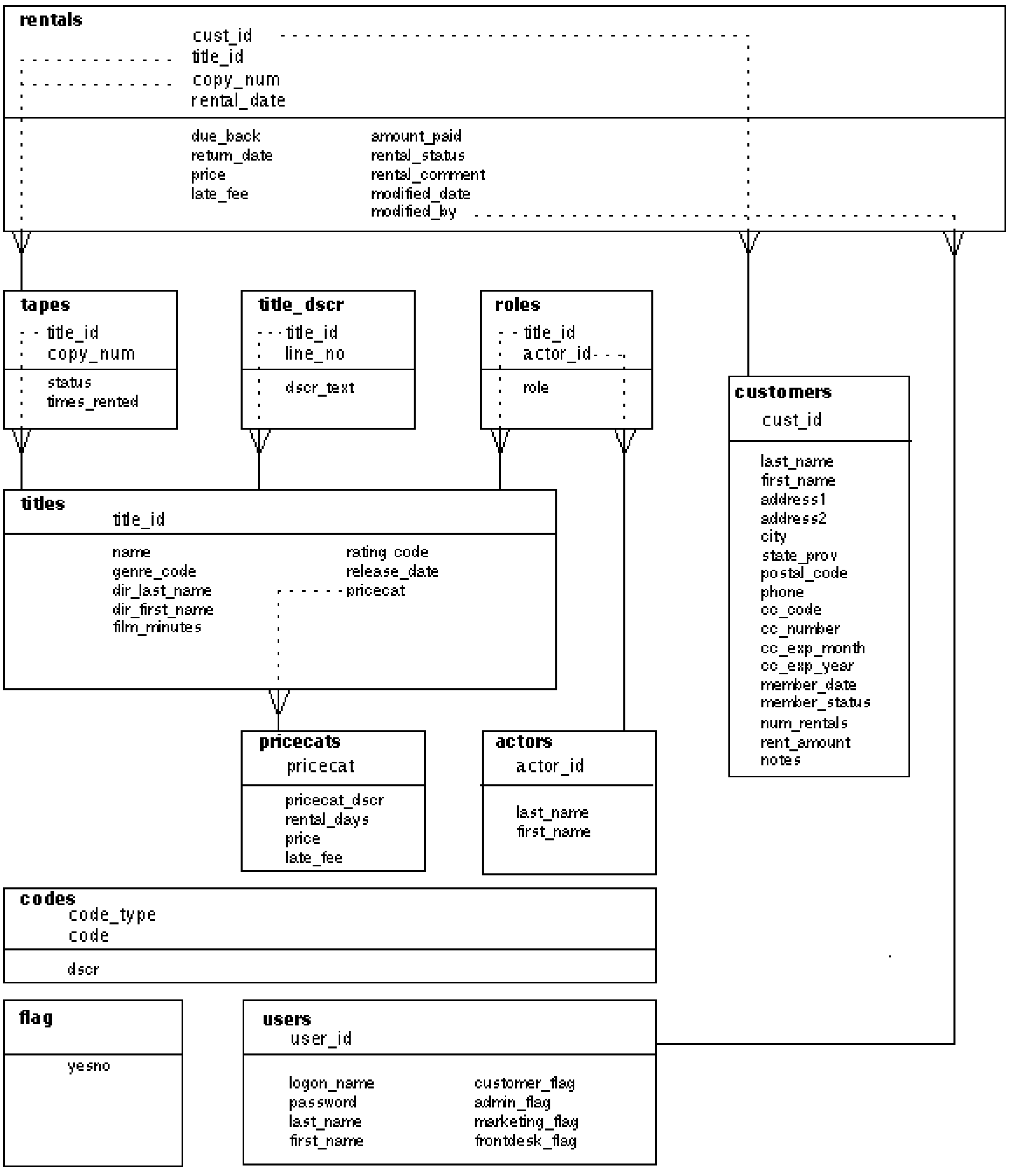


Figure 2-2   Diagram of the videobiz database.

CHAPTER

3 Introduction to SQL

SQL (Structured Query Language) is the database procedure language used by relational database management systems. It was developed by IBM in the early 1970s and then adapted by other software vendors. The American National Standards Institute (ANSI) issued a standard for SQL in 1986 and again in 1992. Although this standard defines a basic set of features that is common to all versions of SQL, each vendor also includes some extensions to SQL in their database products; these extensions are implemented differently.

The scope of SQL gives you complete control over your database operations. There are commands for database definition:

- `CREATE DATABASE`—Creates a database
- `CREATE TABLE`—Adds a table to the database
- `DROP TABLE`—Deletes a table from the database

There are also commands to access and update the data:

- `SELECT`—Retrieves information from the database
- `INSERT`—Adds information to the database
- `UPDATE`—Updates information in the database
- `DELETE`—Deletes information from the database

This chapter contains instructions for using these SQL commands in order to retrieve information from an existing database and to update the database information.

SQL Statements

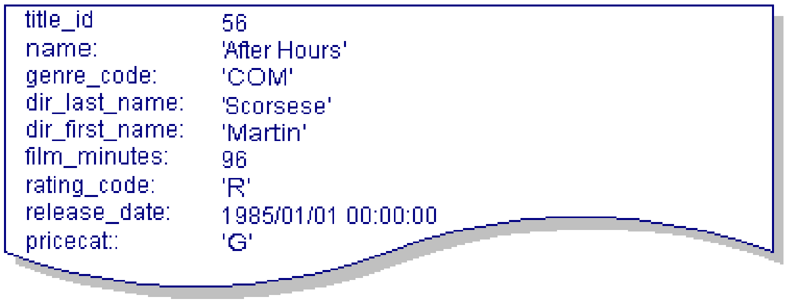
In SQL, commands are called statements and consist of one or more keywords followed by various expressions and clauses. The keywords, used at the beginning of the statement, describe the major function of the statement. In addition to the keywords, most statements also reference at least one database table by name. Since the table is the main storage container for information in a relational database, the tables to be accessed are included in some clause of the statement.

SELECT Statement

The `SELECT` statement retrieves information from database tables and returns it to you in the form of query results.

```
SELECT * FROM titles;
```

In this example, the `FROM` clause lists the database tables from which data will be retrieved. The * tells the database to fetch all of the columns from the tables specified in the `FROM` clause. Therefore, the statement selects all the columns and all the rows from the database table titles.

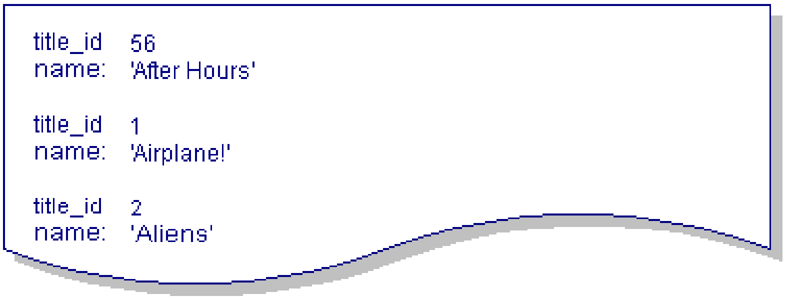


```
title_id          56
name:             'After Hours'
genre_code:       'COM'
dir_last_name:    'Scorsese'
dir_first_name:   'Martin'
film_minutes:     96
rating_code:      'R'
release_date:     1985/01/01 00:00:00
pricecat::        'G'
```

The collection of rows retrieved from the database is called a result set.

To fetch data from just some of the columns as opposed to all columns, replace the asterisk (*) with a list of column names, each name separated from the next by a comma. This is called the select list.

```
SELECT title_id, name FROM titles;
```



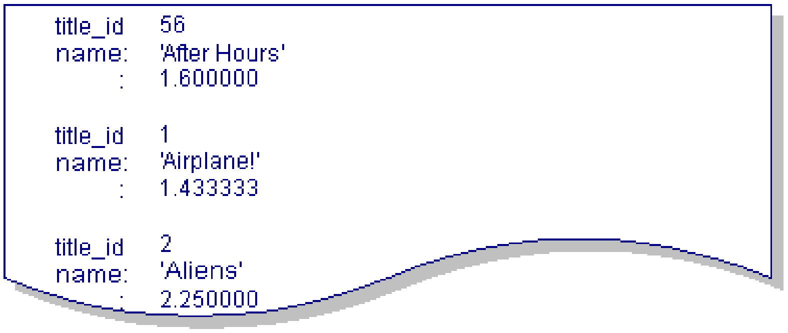
```
title_id   56
name:     'After Hours'

title_id   1
name:     'Airplane!'

title_id   2
name:     'Aliens'
```

Generally, the select list consists of a list of items separated by commas. The select list can include:

- Column names associated with the table or tables in the `FROM` clause.
- An expression which can be a constant, column name, function, subquery, or any combination of these connected by arithmetic and bitwise operators.

The following statement uses the arithmetic operator / to calculate the running time of the video in hours instead of minutes:

```
SELECT title_id, name, film_minutes / 60 FROM titles;
```



```
title_id   56
name:     'After Hours'
       :   1.600000

title_id   1
name:     'Airplane!'
       :   1.433333

title_id   2
name:     'Aliens'
       :   2.250000
```

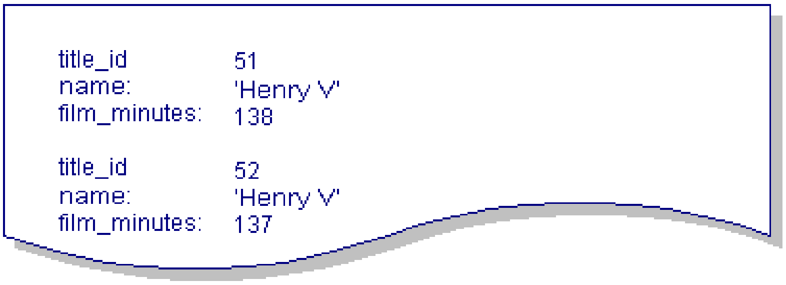
SQL also provides aggregate functions that compute sums, minimum values, and other such operations over all selected rows. The following statement uses the aggregate function `COUNT` to determine the number of rows in the titles table.

```
SELECT COUNT(title_id) FROM titles;
```

WHERE Clause

To select specific rows in a database table, add a `WHERE` clause to the `SELECT` statement:

```
SELECT title_id, name, film_minutes FROM titles
   WHERE name = 'Henry V';
```



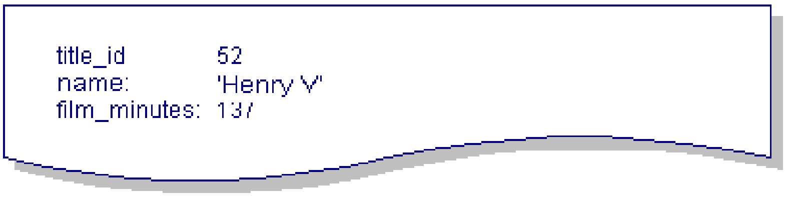
```
title_id        51
name:           'Henry V'
film_minutes:   138

title_id        52
name:           'Henry V'
film_minutes:   137
```

The `WHERE` clause ensures that the result set includes only certain rows of data. In this example, the results include information for all the videos having the name Henry V.

The `WHERE` clause can also specify a search condition to further refine the query. For example, the following `SELECT` statement uses the logical operator `AND`.

```
SELECT title_id, name, film_minutes FROM titles
   WHERE name = 'Henry V' AND dir_last_name = 'Olivier';
```

The result set contains the information for the version of Henry V directed by Sir Laurence Olivier.



```
title_id        52
name:           'Henry V'
film_minutes:   137
```

There are a variety of search conditions and qualifying clauses which let you refine your database queries. For example, they retrieve:

- Rows with column values in a certain range (`BETWEEN`)
- Rows containing a certain pattern (`LIKE`)
- Rows in ascending or descending order (`ASC`, `DESC`)

- Rows meeting search conditions listed in a subquery (`IN`, `EXISTS`)
- Summary values on specific columns (Aggregate Functions)

Refer to Chapter 6, “SQL Reference,” for detailed information on each of these SQL elements.

UPDATE Statement

The `UPDATE` statement lets you to modify values in columns belonging to a specified database table.

```
UPDATE titles SET pricecat = 'G' WHERE title_id = 62;
```

The first keyword, `UPDATE`, defines the purpose of the statement. It is followed by the table name. The `SET` clause specifies which columns are to be updated and defines the new values for those columns. The `WHERE` clause specifies which rows are to be updated.

WHERE Clause

If a `WHERE` clause is not included in the `UPDATE` statement, every row in the table gets updated. The `WHERE` clause must include all the column specifications you need in order to uniquely identify the rows to be updated. The following statement, without the copy number, updates all copies of this video title in the tapes table:

```
UPDATE tapes SET status = 'A' WHERE title_id = 62;
```

To update a single row in the tapes table, you need to provide the primary key, which in tapes table is a combination of the `title_id` and `copy_num` columns.

```
UPDATE tapes SET status = 'A' WHERE title_id = 62 AND copy_num = 2;
```

INSERT Statement

The `INSERT` statement adds new rows of data to a database table. The syntax can vary depending on whether you insert a value into every column or only into selected columns in the table.

```
INSERT INTO tapes VALUES (62, 2, 'A', 0);
```

This statement inserts a new row to the tapes table; the `VALUES` clause specifies the data values for every column in the table. An `INSERT` statement of this type, must provide values for every column in the order in used by the database table. Or, you can include column list in the statement which specifies which columns should receive values and in which order. Therefore, you are not dependent on knowing how the columns are listed in a particular database table. The following statement includes a column list and the results are identical to the previous `INSERT` statement:

```
INSERT INTO tapes
   (title_id, copy_num, status, times_rented)
   VALUES (62, 2, 'A', 0);
```

With a column list, you do not have to enter a column value for each column. You can list only the columns where you plan to make an entry.

```
INSERT INTO tapes (title_id, copy_num, status) VALUES (62, 2,
'A');
```

You can also update the columns in any order; however, the order of the column list and the order of the value list must match.

```
INSERT INTO tapes (status, title_id, copy_num) VALUES ('A', 62,
2);
```

DELETE Statement

The `DELETE` statement removes selected rows of data from a database table. The `FROM` clause names the table to be modified. The `WHERE` clause specifies exactly which rows of the table are to be deleted. If you do not include a `WHERE` clause, the `DELETE` statement deletes every row in the table.

```
DELETE FROM titles WHERE title_id = 62;
```

The `WHERE` clause needs to include all the necessary column specifications to ensure that only certain rows are deleted. To delete a single row, use the row's primary key to identify the row; remember that a primary key can be a combination of more than one column.

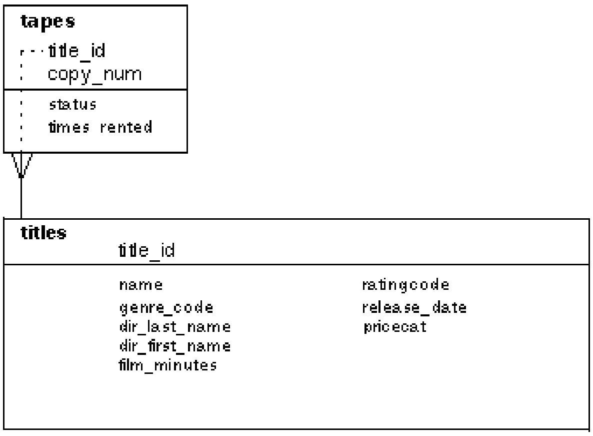
The following statement deletes a single row from the tapes table by specifying values for both the `title_id` and `copy_num` columns.

```
DELETE FROM tapes WHERE title_id = 62AND copy_num = 2;
```

SQL Concepts

Multi-Table Queries

The ability to select data from two or more database tables is one of the great advantages of a relational database. SQL lets you compare any pair or pairs of columns from two or more tables by matching the contents of the related columns.



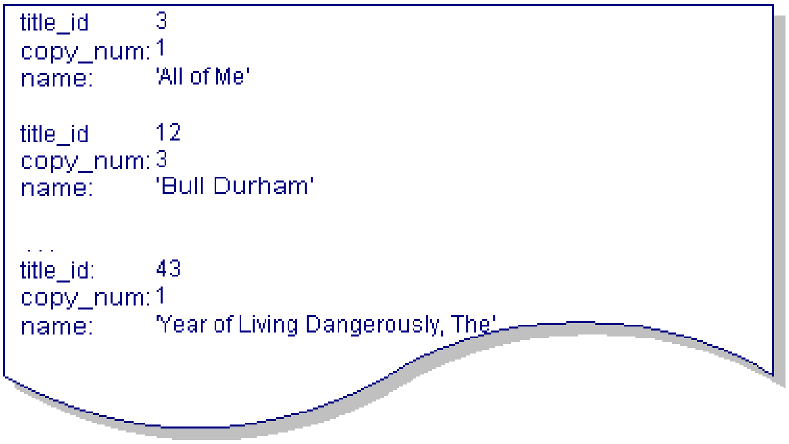
You can join the two tables, tapes and titles, by equating the `title_id` column in the titles table with the `title_id` column in the tapes table. The `WHERE` clause specifies the relationship.

```
SELECT * FROM titles, tapes
   WHERE titles.title_id = tapes.title_id;
```

The following `SELECT` statement joins two tables and the result set provides you with a list of videos that are available (have status = A) for rental.

```
SELECT tapes.title_id, tapes.copy_num, titles.name

FROM titles, tapes
   WHERE titles.title_id = tapes.title_id
```

```
    AND tapes.status = 'A';
```



```
title_id      3
copy_num: 1
name:         'All of Me'

title_id      12
copy_num: 3
name:         'Bull Durham'

 . . .
title_id:     43
copy_num: 1
name:         'Year of Living Dangerously, The'
```

For more information on different types of joins and on joining multiple tables, refer to page 6-27.

Correlation Names

When using joins, instead of using the entire table name throughout the statement, you can give the table a correlation name. The following example uses a as the correlation name for the titles table and b as the correlation name for the tapes table.

```
SELECT b.title_id, b.copy_num, a.name
    FROM titles a, tapes b
    WHERE a.title_id = b.title_id
    AND b.status = 'A';
```

You can also use correlation names to perform a self-join, which joins a table to itself so that you can compare values in the same column.

Correlation names must follow the naming conventions for identifiers. They can be 31 characters in length and include letters, numbers, and underscores.

Aggregate Functions

Aggregate functions calculate different types of summary information on rows in a database table including:

- Sums of numeric columns.
- Average, maximum and minimum values in columns.
- Number of rows containing a specific column value.

For more information on aggregate functions, refer to page 6-4.

Transactions

Transactions are units of work on a database. A transaction consists of a series of database statements to be completed as a unit. If the unit is unable to be completed, the statements can be rolled back, restoring the database to its prior state before the transaction started. This ensures the integrity of the database. For example, in the videobiz database, each new entry in the titles table also needs entries in the tapes table, and possibly in the actors and roles tables. All these entries could be grouped into one transaction so that you know the entry is complete.

Implementation in a JDB Database

Database engines implement transactions differently. In JDB, after you declare a connection, a transaction automatically starts on that connection. Additional transactions can then be defined using commit and rollback commands.

The commit command saves the changes to the database. The rollback command undoes any changes made to the database since the start of the transaction. The execution of either commit or rollback starts a new transaction.

Using the videobiz database example, a customer comes to the front desk to rent a video. When the clerk checks out the video, a transaction is started to perform the following actions:

- Insert the rental information into the rentals table.
- Update information about the customer's rentals in the customers table.
- Update information about the video tape status in the tapes table.

If any of the statements fail, the transaction must be rolled back. If all of the statements execute without any errors, the transaction can be committed.

JDB performs an automatic commit when you leave an isql or JISQL session. You must issue a rollback command if you do not want to save your database changes.

The following statements from a JISQL script illustrate the sample rental transaction which rents a video to a customer:

```
insert into rentals
   (cust_id, title_id, copy_num, rental_date, due_back,
   return_date, price, late_fee, amount_paid, rental_status,
   rental_comment, modified_date, modified_by)
   values
   (3, 69, 2, '1993/10/29 19:56:00', '1993/11/01 00:00:00',
   NULL, 3.50, 1.00, 3.50, 'C', NULL, '1993/10/29 19:56:00',
   'jenny');

update customers set num_rentals = 75, rent_amount = 201.50
   where cust_id = 3;

update tapes set status = 'O' where title_id = 69
   and copy_num = 2;

$COMMIT;
```

Executing SQL

In Panther, you can execute SQL commands:

- By using JISQL
- Via JPL procedures
- Through the transaction manager

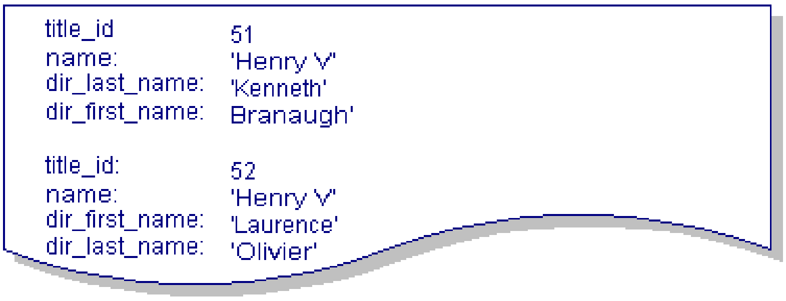
The examples in this book use the JISQL syntax unless otherwise indicated.

Using JISQL

To use JISQL, start the JISQL utility and connect to a database. In the SQL scripting area, enter the text of the SQL statement followed by the termination character, a semicolon (;), to end the statement. Any date values or character strings must be enclosed in single quotation marks. For example:

```
SELECT title_id, name, dir_first_name, dir_last_name

    FROM titles

    WHERE name = 'Henry V';
```

retrieves the following rows:



```
title_id           51
name:              'Henry V'
dir_last_name:     'Kenneth'
dir_first_name:    Branaugh'

title_id:          52
name:              'Henry V'
dir_first_name:    'Laurence'
dir_last_name:     'Olivier'
```

Refer to Chapter 5, “Using JISQL,” for more information on using the JISQL utility.

Using JPL

The same SQL command in a JPL procedure named `query1` would look like this:

```
proc query1
dbms query SELECT title_id, name, dir_first_name, \
    dir_last_name FROM titles \
    WHERE name = 'Henry V'
return 0
```

The JPL continuation character (\) is needed whenever a command is not completed on one line. A termination character is not required at the end of the statement since it is added automatically by Panther's database interface to JDB.

See Also

For more information on writing JPL, refer to page 19-1 in the *Developer's Guide*.

For information on mapping data to Panther variables, refer to page 29-3 in the *Developer's Guide*.

CHAPTER

4 Database Elements

This chapter describes JDB-specific elements, for example, the database and database column naming conventions and data types that are supported in a JDB database. The information about your JDB database and tables, such as its primary and foreign key specification, are stored in system tables which are automatically created.

Also discussed are:

- Journal files which store information about database activities.
- Configuration variables for setting up the JISQL (or isql) environment.

Naming Conventions

Databases

Each database is stored as an operating system file. Therefore, the name for the database must follow operating system naming conventions. If the database name contains characters that are not alphanumeric, the name specified in the `CREATE DATABASE` statement must be enclosed in single quotation marks.

You can create databases when you are connected to `@system` or when you are connected to another database. For more information on the `CREATE DATABASE` statement, refer to page 6-9.

Identifiers

Identifiers, such as table names and column names, must start with a letter, but they can contain letters, numbers, and underscores in any combination. They cannot contain dollar signs or periods. The maximum length of an identifier is 31 characters. If more than 31 characters are entered, the value is truncated.

Table names must be unique within the database. Each column name in a database table must be unique within that table.

Since column names can be duplicated in different tables, therefore, some statements might require that you uniquely identify a column name by including its database table name. For example, in the sample database, the `last_name` column appears in more than one table. To specify the `last_name` column in the actors table, use the following syntax:actors.`last_name`

You cannot use any of the JDB keywords as an identifier. For a list of the keywords, refer to page C-1.

JDB is case insensitive and stores the identifiers in lower case regardless of which case is used to enter them. If you enter `address1`, `ADDRESS1`, or `Address1`, JDB stores the column as address1.

Data Types

Table 4-1 lists the data types available in JDB:

Table 4-1 Data types in JDB

Data Type	Description
`INT`	Numeric value (stored as `LONG`)
`LONG`	Numeric value
`FLOAT`	Numeric value

Table 4-1 Data types in JDB *(Continued)*

Data Type	Description
`DOUBLE`	Numeric value
`DATETIME`	Date and time value–format `yyyy/mm/dd hh:mm:ss`
`CHAR`	Character string

For more information on data types, refer to page 6-15.

System Tables

When you create a new database, five system tables are automatically created that contain information (such as column names and primary/foreign key specifications) about the database itself. You can query for information stored in these tables just like any other database table; however, you can not edit these tables.

The systabs system table contains information about each database table.

Table 4-2 Columns belonging to systabs system table

Column Name	Description
tname	Table name
ttype	Table type
ncols	Number of columns
seek	Column for internal use

The syscols system table contains information about the columns in each database table.

Table 4-3 Columns belonging to syscols system table

Column Name	Description
tname	Table name
cname	Column name

ctype	Column type–numeric values correspond to the following data types:
	`101 INT` (stored as `LONG` in the current release)
	`102 LONG`
	`103 FLOAT`
	`104 DOUBLE`
	`105 DATETIME`
	`106 CHAR`
	`1125 INT, NOT NULL (stored as LONG, NOT NULL)`
	`1126 LONG, NOT NULL`
	`1127 FLOAT, NOT NULL`
	`1128 DOUBLE, NOT NULL`
	`1129 DATETIME, NOT NULL`
	`1130 CHAR, NOT NULL`

length	Column length

The syskeys system table specifies the primary and foreign keys.

Table 4-4 Columns belonging to syskeys system table

Column Name	Description
tname	Table name
keyno	Number assigned to the key column of the table–primary key is always 1
resolved	Column for internal use
hasreflist	Indicator specifying whether a reference list is included in the `REFERENCES` clause of the `CREATE TABLE` statement

Table 4-4 Columns belonging to syskeys system table *(Continued)*

Column Name	Description
rtname	Name of the database table specified in the `REFERENCES` clause of the `CREATE TABLE` statement
keytype	Indicator specifying primary key (P), foreign key (F), or unique entry (U)

The syskeycols system table contains information about each primary and foreign key column.

Table 4-5 Columns belonging to syskeycols system table

Column Name	Description
tname	Table name
keyno	Number assigned to the key column in `syskeys`
position	Order of the column in a composite key, if applicable
cname	Column name

The `sysrkeycols` system table contains information about the columns listed in the `REFERENCES` clause of a `CREATE TABLE` statement.

Table 4-6 Columns belonging to sysrkeycols system table

Column Name	Description
tname	Table name
keyno	Number assigned to the key column in syskeys
position	Order of the column in a composite key, if applicable
cname	Column name

Journal Files

JDB automatically creates journal files in your database directory. These files record your actions on the current database. The current journal file is named `j1databaseName`. When you start a JDB session, the current journal file is copied to a file named `j0databaseName`. If the file `j0databaseName` already exists, its contents are replaced. Journal files can be reinstated using the utility `jdbroll`.

Configuration

Specifying an Editor

The environment variables `SMEDITOR` or `EDITOR` determine which text editor is available in `JISQL` or in `isql`. When using `JISQL`, the specified editor can be used to make changes to the SQL text window. When using isql, entering the edit command displays the last statement in the specified text editor.

Error Messages

The error messages for JDB are stored in the Panther message file. If the program has trouble locating the error messages, check the setting of the variable `SMVARS`.

Connecting to a JDB Database

If you place your JDB database in the application directory or in one of the directories listed in `SMPATH`, you do not need to specify the path in the `DBMS DECLARE CONNECTION` statement.

CHAPTER

5 Using JISQL

JISQL is a graphical tool you can use to:

- Create a JDB database
- Create database tables for a new or existing JDB database
- Display table definitions for the current database
- Write and execute interactive SQL scripts for use with your JDB databases

Starting JISQL

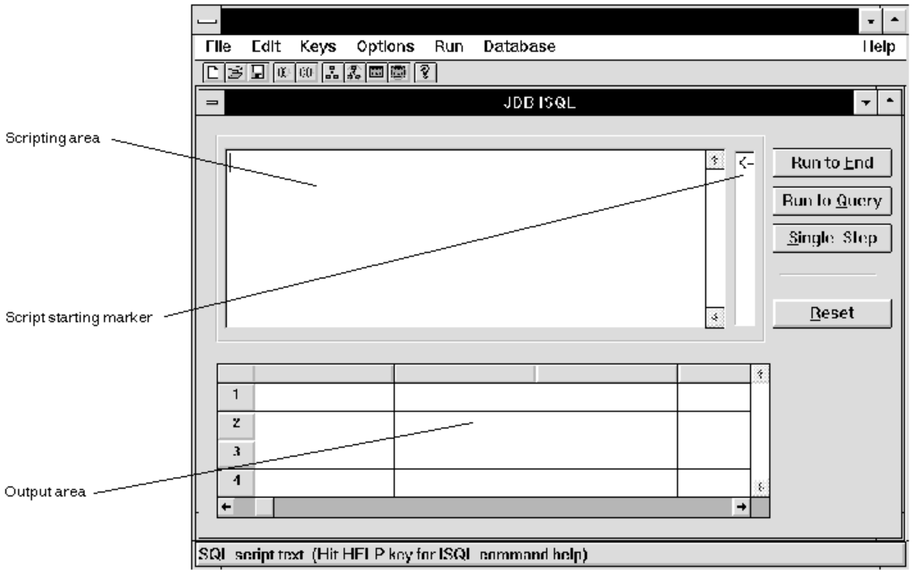
To start JISQL, do either of the following:

- At the command line, type:

  `$SMBASE/util/jisql` (under UNIX)

  `$SMBASE\util\jisql` (under Windows)
- Under Windows, choose the JDB ISQL icon.

The JDB ISQL window opens.



The JDB ISQL window provides an area into which you enter a SQL script. Menu options allow you to:

- Execute SQL commands, either from the scripting area or from a file.
- Select options for executing the SQL script.
- Connect to a JDB database.
- Create a JDB database.
- Drop a JDB database.

To exit JISQL, choose File⟶Exit.

JDB performs an automatic `COMMIT` when you leave a JISQL session. Issue a `$ROLLBACK` macro command if you do not want to save your database changes.

JDB Database Connections

Before you can create or view database tables or perform any other database operations, you must connect to the database. (If your SQL script includes a command to connect to the database, you need not connect as described here before executing the script.)

To connect to an existing database:

1. Choose Database⟶Connect. A file selection dialog box opens.
2. Specify the name of the database, and choose OK. The file selection dialog box closes, and you return to the JDB ISQL window.

To connect to a new database:

When you create a new database, you must connect to it before defining any of its tables.

- In the Create Database window, specify that you want to automatically connect to the new database at the time it is created.

Refer to page 5-4 for instructions on creating and connecting to a new database.

To disconnect from the current database:

- Choose Database⟶Disconnect.

  JDB performs an automatic `COMMIT` when you close a database connection. Issue a `$ROLLBACK` macro command if you do not want to save your database changes.

You can be connected to only one database at a time. If you connect to a database while a previous connection is still current, JISQL automatically disconnects from the first database before connecting to the next one.

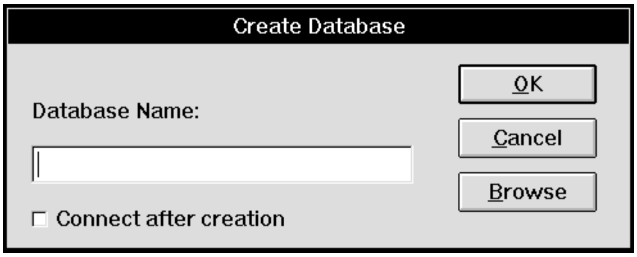
Executing Operating System Commands from JISQL

To execute an operating system command from JISQL:

1. Choose Options→System Command. A dialog box opens with a field for you to enter the command.
2. Enter the desired system command, and choose OK. The command is executed; display output is platform-dependent.
3. Depending on the platform, if you are not returned to the JDB ISQL window when the command has finished executing, press any key.

Creating a New Database

1. From the JDB ISQL window, choose Database→Create Database. The Create Database dialog opens.



2. Enter the name of the database you want to create. You can choose the Browse push button to view the names of existing files from a file browse dialog box. When you have finished with this dialog box, choose OK to return to the Create Database window.
3. (Optional) Select the Connect After Creation check box to automatically connect to this database after it is created in order to create tables and enter data.

4. Choose OK.

   If you selected the Connect After Creation check box, the status line message confirms that you are connected to the database. If you did not select this check box, the message indicates only that the database was successfully created.

Creating Database Tables

Use the JISQL graphical interface to add tables to a newly created database or to an existing database.

To create a database table:

1. Connect to the applicable database.
2. Choose Database⟶Create Table. The Create Table dialog box opens.

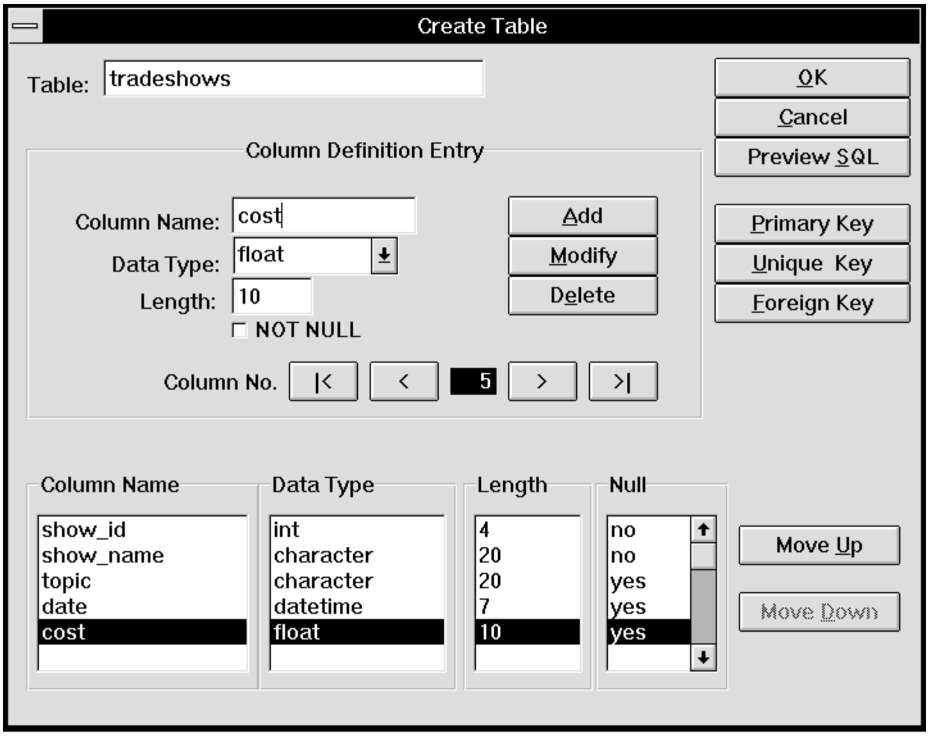


Table: tradeshows

Column Name: cost
Data Type: float
Length: 10
☐ NOT NULL
Column No. 5

Column Name	Data Type	Length	Null
show_id	int	4	no
show_name	character	20	no
topic	character	20	yes
date	datetime	7	yes
cost	float	10	yes

3. Enter the table name in the Table field

4. Define each column, one at a time, in the Column Definition Entry area. Refer to page 5-7 for a more detailed explanation of column definition.

5. Specify the keys for this table. Refer to page 5-8 for information on specifying primary, unique, and foreign keys.

6. (Optional) Choose the Preview SQL push button to display the SQL command that JISQL will generate to create the table, as it is currently defined.
When you have finished reviewing the SQL command, choose Done to resume in the Create Table window.

7. Choose OK to create the table you have just defined. A message is displayed confirming that the table has been created.

To populate the table, create and run a SQL script containing the applicable `INSERT` statements. For information on entering and running SQL scripts under JISQL, refer to page 5-18.

Defining Columns in a Database Table

The Column Definition Entry area of the Create Table window allows you to add, modify, or delete columns in the database table you are creating. In addition, the Create Table window provides push buttons that enable you to rearrange the columns in the table.

To add a new column to the table you are creating:

1. In the Column Definition Entry area, specify the column name and data type. For some data types, you must also specify the length.
2. Select the `NOT NULL` check box if null values are not to be permitted in this column.

   Note: `NULL` values are not permitted in primary key columns.
3. Choose Add. Once the column is added, its position in the table is shown in the middle portion of the Column No. field. It is also added to the column summary for the table, displayed in the lower portion of the Create Table window.
4. Repeat the preceding steps for each column you want to define for the table.

To change or delete a column's definition:

You can modify or delete a column at any point prior to completing the table definition.

1. Specify the applicable column by doing either of the following:
   - Select its entry in the summary area of the Create Table window.
   - Choose the Column No. up/down indicators in the Column Definition Entry area; continue choosing the appropriate indicator until the desired column definition is displayed. The column number of the current column is shown in the middle of the Column No. field. Click on < to display the

previous column, or > to display the next column. Click on |< to display the first column in the table, or >| to display the last.

The definition for the specified column is shown in the Column Definition Entry area.

2. Change any of the column definition parameters as desired.

   Note: You cannot remove `NOT NULL` from a primary key column.

3. Do either of the following, depending on the desired results:
   - Choose Modify to change the column definition.
   - Choose Delete to remove the column from the table.

To change the order of columns in the table:

Select a column in the summary area of the Create Table window. Choose the Move Up or Move Down push button to move it one place up or down. Continue until the column is in the desired location.

Defining Keys for a Database Table

Push buttons in the Create Table window allow you to define primary, unique, and foreign keys into the table. Refer to page 2-4 for an explanation of primary keys. Refer to page 2-6 for an explanation of foreign keys.

1. Define all columns that will be keys into the table. If you are defining foreign keys, the referenced table must have been created previously.

2. Choose the applicable push button: Primary Key, Unique Key, or Foreign Key. The corresponding key definition window opens.

3. Create, modify, or delete the applicable key definition. Refer to page 5-9 for instructions on using the Primary and Unique Key Definition windows. Refer to page 5-11 for instructions on using the Foreign Key Definition window.

4. When you are done with the key definition, choose OK. You return to the Create Table window.

   If any column required for a key was not defined as `NOT NULL` when it was created, JISQL makes the necessary change to the column definition and displays an appropriate message. Acknowledge the message by choosing OK.

5. Continue creating, modifying, and deleting keys for the table as needed. You can create a new key or modify or delete an existing key at any point prior to completing the table definition.

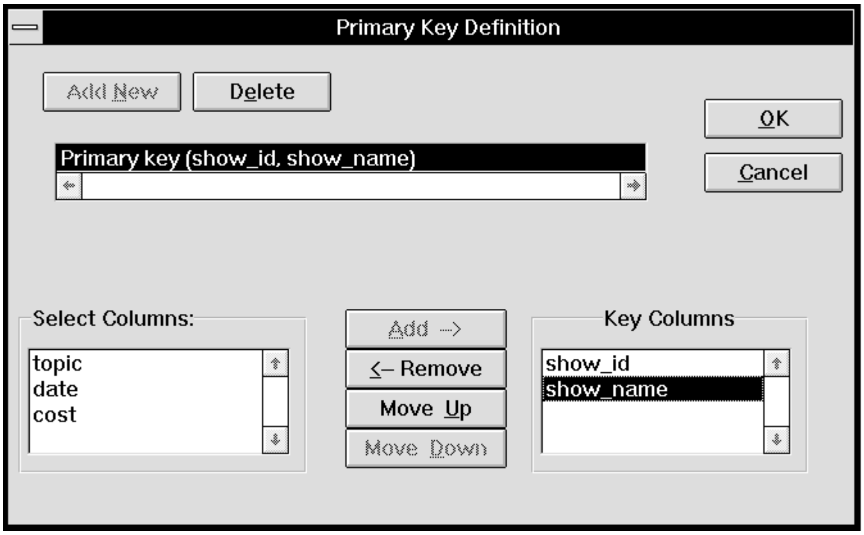
Primary Key and Unique Keys

The Primary Key and Unique Key Definition windows are similar in appearance and function. Each consists of:

- A text area showing the SQL definition generated for each key defined on the screen. When you want to modify or delete an existing key, select it from this area. As you create or modify a key definition, its SQL text area is updated to reflect changes.
- Push buttons (Add New and Delete) to specify that you want to add a new key or delete an existing one.
- A Select Columns area listing the table columns not used in the selected key.
- A Key Columns area listing, in order, the table columns belonging to the selected key.
- Push buttons (Add⟶ and ⟵Remove) to add a selected column in the Select Columns area to the key and to remove a selected column in the Key Columns area from the key.
- Push buttons (Move Up and Move Down) to rearrange the order of columns in the selected key.

From the Create Table window, choose the Primary Key button to open the Primary Key Definition window, or choose Unique Key to open the Unique Key Definition window.

When you are finished working in the Primary Key or Unique Key Definition window, choose OK to save your changes and return to the Create Table window; or choose Cancel to return without saving your changes.



Once the applicable key definition window is open, you can add, modify, or delete keys as follows:

To add a new primary key or unique key:

1. On the Primary Key or Unique Key Definition window (as applicable), choose Add New. All the table columns are listed in the Select Columns area. If a primary key is already defined for the table, the Add New push button is not available, since only one primary key statement is permitted. Either delete the existing key or modify it.

2. For each column you want in the key, select the column from the Select Columns area and choose Add⟶. The column name is removed from the Select Columns area and appears in the Key Columns area.

3. To change the order of a column in the key, select it in the Key Columns area and choose the Move Up or Move Down push button to move it to the desired location.

To modify an existing primary key or unique key:

1. On the Primary Key or Unique Key Definition window (as applicable), select the SQL definition corresponding to the key you want to modify. The Select Columns and Key Columns areas reflect the current definition of the key.
2. Select the column you want to add to the key from the Select Columns area and choose Add⟶. The column name is removed from the Select Columns area and appears in the Key Columns area.
3. Select the column you want to remove from the key from the Key Columns area and choose ⟵Remove. The column name is removed from the Key Columns area and appears in the Select Columns area.
4. To change the order of a column in the key, select it in the Key Columns area and choose the Move Up or Move Down push button to move it to the desired location.

To delete an existing primary or unique key:

1. On the Primary Key or Unique Key Definition window (as applicable), select the SQL definition corresponding to the key you want to delete. The Select Columns and Key Columns areas reflect the current definition of the key.
2. Choose Delete. The SQL definition for this key is deleted from the text area, and the Select Columns and Key Columns areas are emptied.

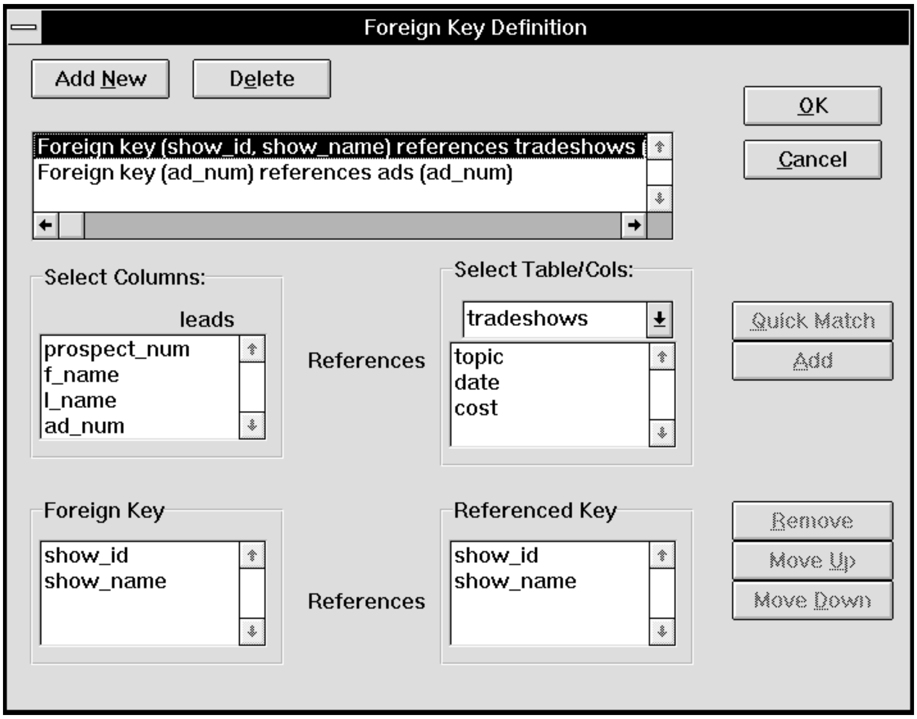
Foreign Keys

The Foreign Key Definition window consists of:

- A text area showing the SQL definition that will be generated for each key defined on the screen. When you want to modify or delete an existing key, you select it from this area. As you create or modify a key definition, the SQL text area is updated to reflect any changes.
- Push buttons (Add New and Delete) to specify that you want to add a new key or delete an existing one.
- A Select Columns area listing the table columns not used in the selected key.

- A Select Table/Cols area with:
  - An option menu for you to choose the referenced table.
  - A listing of the columns in the chosen table not used in the selected key.
- Push buttons (Add and Quick Match) to add a selected column in the Select Columns area and its foreign table column reference to the key. The Quick Match button allows you to reference all columns in the current table to identically-named primary key columns in the chosen table without having to explicitly choose any columns from the lists.
- A Foreign Key area listing the table columns used in the selected key. Each column name in this area is lined up beside the corresponding column in the Referenced Key area.
- A Referenced Key area listing each column in the chosen table that is referenced in the selected key. Each column name in this area is lined up beside the corresponding column in the Foreign Key area.
- Push buttons (Move Up and Move Down) to re-arrange the order of columns in the selected key.
- Push button (Remove) to remove a selected Foreign Key/Referenced Key column pair from the selected key.

From the Create Table window, choose the Foreign Key button to open the Foreign Key Definition window.



Foreign key (show_id, show_name) references tradeshows (
Foreign key (ad_num) references ads (ad_num)

Select Columns: leads		Select Table/Cols: tradeshows
prospect_num	References	topic
f_name		date
l_name		cost
ad_num		

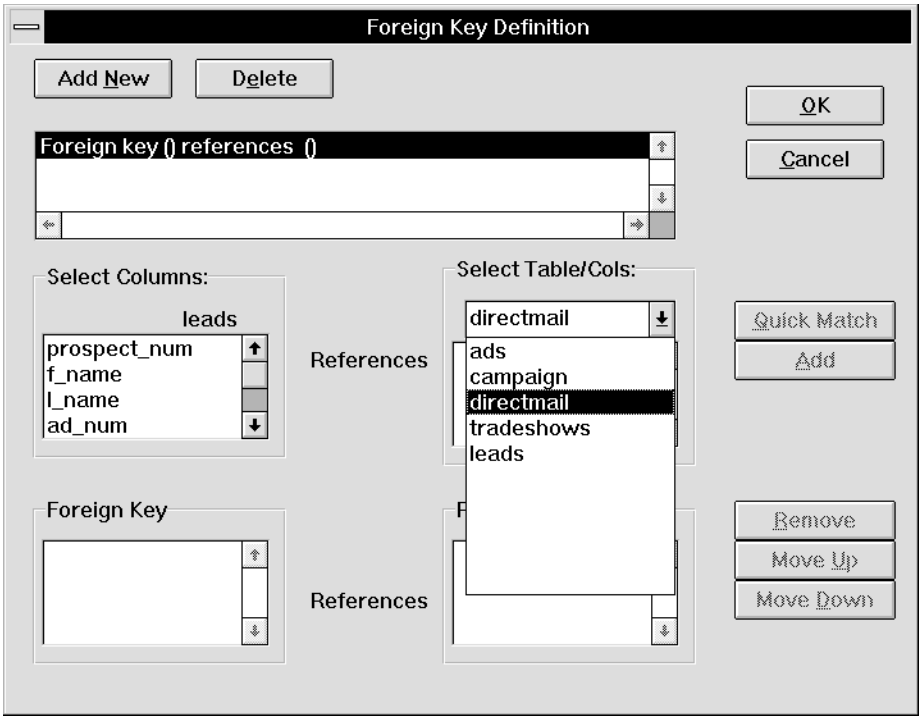
Foreign Key		Referenced Key
show_id	References	show_id
show_name		show_name

When you are finished working in the Foreign Key Definition window, choose OK to save your changes and return to the Create Table window; or choose Cancel to simply return without saving your changes.

When the Foreign Key Definition window is open, you can add, modify, or delete foreign keys as follows:

To add a new foreign key:

1. On the Foreign Key Definition window, choose Add New. All the table columns are listed in the Select Columns area.
2. Select the option menu in the Select Table/Cols area and choose the table to be referenced.



3. Once you have chosen the table, a list of columns in that table is displayed. Reference the columns for the key in one of the following ways:

   - For each column in the foreign key, select the current table column from the Select Columns area and select the foreign column to be referenced from the Select Table/Cols area. Choose Add.

   - Choose Quick Match to reference the selected columns in the current table with identically-named primary key columns in the chosen table.

   The column names are removed from the Select Columns area and the Select Table/Cols area; they appear in the Foreign Key and Referenced Key areas, respectively.

4. To change the order of a Foreign Key/Referenced Key column pair in the key, select either column in the pair and choose the Move Up or Move Down push button to move the pair to the desired location.

To modify an existing foreign key:

1. On the Foreign Key Definition window, select the SQL definition corresponding to the key you want to modify. The Select Columns, Select Table/Cols, Foreign Key, and Referenced Key areas reflect the current definition of the key.
2. For each column you want to add to the foreign key, do either of the following:
   - Select the current table column from the Select Columns area and select the foreign column to be referenced from the Select Table/Cols area. Choose Add.
   - Choose Quick Match to reference columns in the current table to identically-named primary key columns in the chosen table.
3. For each column pair you want to remove from the key, select either column in the pair and choose Remove. The column names are removed from the Foreign Key and Referenced Key areas and appear in the Select Columns and the Select Table/Cols areas, respectively.
4. To change the order of a column in the key, select it in the Key Columns area and choose the Move Up or Move Down push button to move it one place up or down. Continue until the column is in the desired location.

To delete an existing foreign key:

1. On the Foreign Key Definition window, select the SQL definition corresponding to the key you want to delete. The Select Columns, Select Table/Cols, Foreign Key, and Referenced Key areas reflect the current definition of the key.
2. Choose Delete. The SQL definition for this key is deleted from the text area, and the Select Columns, Select Table/Cols, Foreign Key, and Referenced Key areas are emptied.

Maintaining a Database

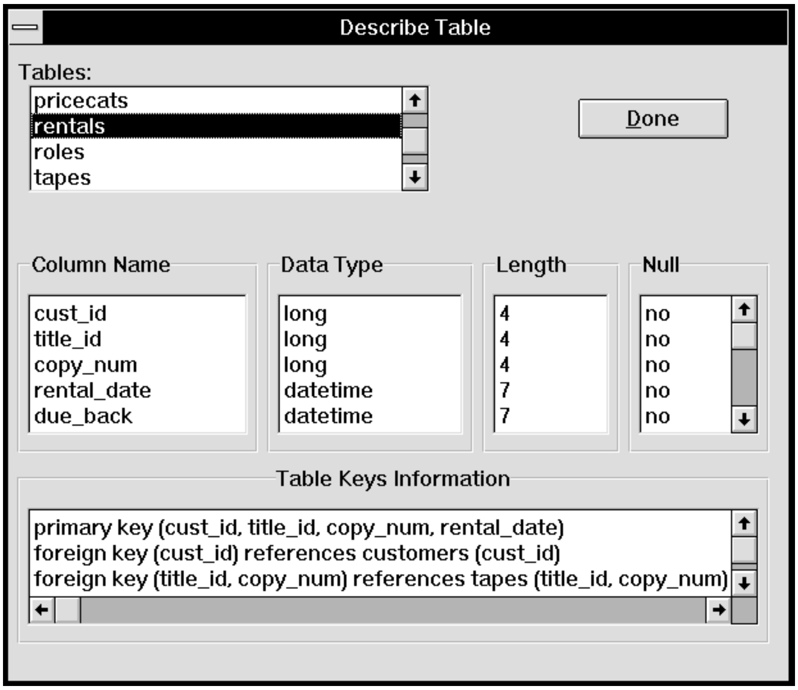
With JISQL, you can perform the following database maintenance functions without having to write SQL code:

- Display the definition of any or all tables in the current database.
- Drop a database.
- Drop specified tables from a database.

Notes: To perform database maintenance operations involving the data itself, such as populating tables, viewing data, etc., you must explicitly write and execute the required SQL statements. Refer to page 5-18 for information on running SQL interactively using JISQL.

Displaying Database and Table Definitions

1. Connect to the database whose definitions you want to display.
2. Choose Database⟶Describe. The Describe Table window opens, displaying a list of all the tables in the database.
3. Select the table whose definition you want to display. The column definitions and key information for this table are displayed. Continue in this way to display table definitions one at a time.



Column Name	Data Type	Length	Null
cust_id	long	4	no
title_id	long	4	no
copy_num	long	4	no
rental_date	datetime	7	no
due_back	datetime	7	no

```
primary key (cust_id, title_id, copy_num, rental_date)
foreign key (cust_id) references customers (cust_id)
foreign key (title_id, copy_num) references tapes (title_id, copy_num)
```

4. Choose Done when you are finished viewing table definitions for this database. You return to the JDB ISQL window.

Dropping Tables

1. Connect to the database from which you want to drop a table.
2. Choose Database⟶Drop Table. The Drop Table window opens.
3. Select the applicable table from the drop-down list for the Table Name field.
4. Choose OK. A message is displayed confirming that the table has been dropped.

Dropping a Database

1. Make sure that you are not connected to the database you want to drop.

2. Choose Database→Drop Database. The Drop Database window opens.
3. Enter the database name, or choose the Browse push button to select the database from a file selection dialog box.
4. With the database to be dropped specified in the Database Name field, choose OK. A message is displayed confirming that the database has been dropped.

Running SQL Interactively

Using JISQL, you can run SQL commands either by entering them into the onscreen scripting area or by specifying an ASCII file that contains the desired SQL script. In addition, when you create a SQL script in JISQL, and then save it to a file for future use.

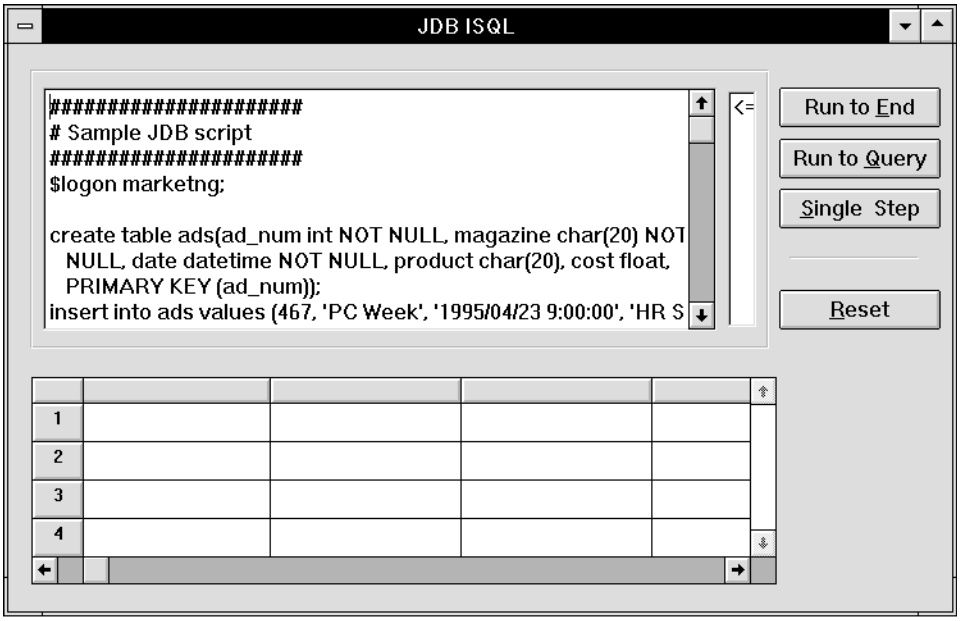
Under JISQL, you can execute any SQL statement that is available in JDB. Refer to page 6-1 for a detailed description of the SQL commands that can be used with a JDB database.

Your SQL script can also contain JISQL macro commands. These macros simplify transaction processing and database maintenance. Refer to page 5-21 for a complete description of the JISQL macros.

JISQL runtime options enable you to control the execution and output of your SQL script. Refer to page 5-22 for a description of the available options and commands.

Writing SQL Scripts

The JDB ISQL window contains an area for entering and editing your SQL script.



```
########################
# Sample JDB script
########################
$logon marketng;

create table ads(ad_num int NOT NULL, magazine char(20) NOT
  NULL, date datetime NOT NULL, product char(20), cost float,
  PRIMARY KEY (ad_num));
insert into ads values (467, 'PC Week', '1995/04/23 9:00:00', 'HR S
```

To enter a SQL script:

Either type directly into the scripting area, or read the script in from an ASCII text file.

To read a text file into the scripting area, choose File→Open Script; a file selection dialog box opens for you to specify the file you want to read in. By default, only filenames ending with the `*.sql` extension are listed in the dialog box.

To edit a SQL script:

You can do either of the following:

- Edit directly in the scripting area. Use the editing keys on your keyboard, or choose the desired editing function the Keys menu.
- Use the default editor by choosing File→Editor (or press PF2). The editor specified in the environment variable `SMEDITOR` is invoked (refer to page 2-6 in the *Configuration* for details on specifying the variable).

To save SQL script displayed in the scripting area:

Do either of the following:

- Choose File→Save Script—The displayed script is saved to the file from which it was read, replacing the original contents of that file with the current script.
- Choose File→Save As—Specify the name of new file in the file selection dialog box.

Use a *.sql extension in naming SQL script files, since only files with this extension appear in the file selection dialog box when you choose File→Open script.

To clear the scripting area:

Choose File→New. The scripting area and data output area are cleared in preparation for entering and running a new script.

Script Format and Syntax

SQL scripts to be executed under JISQL can consist of:

- Any SQL statement available for JDB. Refer to page 6-1 for a description of each statement and its syntax.
- Any JISQL macro command. Refer to page 5-21 for information on these macros.
- Comment lines. Any line beginning with a pound sign (#) is treated as a comment.
- Blank lines.

Only one statement is permitted per line. Each SQL statement and JISQL macro command must be terminated with a semicolon (;). A line without a trailing semicolon is concatenated with the next line until the semicolon is reached. Therefore, one statement can span multiple lines.

JISQL Macro Commands

The macros provided in JISQL are listed in Table 11. Each macro begins with a dollar sign ($) and can be typed in either all uppercase or all lowercase, but not in mixed case. Each macro command must be terminated with a semicolon (;).

Table 5-1 JISQL Macro Commands

Command Syntax	Description
`$COMMIT`	Same as `DBMS COMMIT`. Commits a transaction. Data changes pending in the transaction are applied to the database. (JDB performs an automatic `COMMIT` when you leave a JISQL session or close a database connection.)
`$DESCRIBE` *`table-name`*	Displays a `CREATE TABLE` statement equivalent to the definition of the specified table. Example: `$DESCRIBE` titles; Output of this macro can be redirected to a file by choosing Options→Output to File.
`$DUMP` *`table-name`*	Displays a `CREATE TABLE` statement and an `INSERT` statement for each row in the table. Example: `$DUMP` tapes; Output of this macro can be re-directed to a file by choosing Options→Output to File.
`$LOGON` *`database-name`*	Connects to the specified database. Example: `$LOGON videobiz;` Since JISQL allows only one database connection at a time, this macro closes the previous connection, if there is one, before initiating a new connection.
`$ROLLBACK`	Same as `DBMS ROLLBACK`. Backs out a transaction. The database is restored to its state prior to the start of the pending transaction.

Executing SQL Scripts

1. Enter your SQL script into the scripting area. Refer to page 5-18 for instructions on entering and editing SQL scripts.

2. Connect to the database. (Refer to page 5-3 for instructions on connecting to a database.) Omit this step if your script contains the $LOGON macro to perform the connection.

3. Choose the desired execution and output options from the Options menu. All the following options are toggles; select as many as are applicable:

   - Continue After Error—If an error occurs during batch mode execution, JISQL continues execution after you acknowledge the error message. If the option is not selected, execution stops at the statement that caused the error.

   - Output to File—All output from execution of the SQL script is saved to a file. Select sets from `SQL SELECT` statements are directed to the file and are not displayed on the screen. Output from `$DESCRIBE` and `$DUMP` macros is displayed on the screen as well as saved in the file. If the option is not selected, select sets are displayed in the lower portion of the JDB ISQL window. Refer to page 5-23 for more information on capturing and displaying query results.

   - Record in Log—Information about execution of the SQL script is saved in a log file. Refer to page 5-25 for information on creating and viewing the log file.

4. Position the starting marker on the line of your script where you want execution to begin. The starting marker appears to the right of the scroll bar for the scripting area. To move the starting marker, click in the space to the right of the scroll bar, lining up the mouse cursor with the SQL statement you want to execute next. Initially, the starting marker is beside the first line of the script.

Output and Execution Options

Choose one of the following execution commands to execute the ISQL script. The commands are available both as push buttons on the screen and as Run menu options:

■ Run to End—Start batch mode execution from the starting marker. Execution continues to the end of the script unless an error is encountered. The setting of the Continue After Error toggle determines whether execution is terminated at the point of the error or if it continues after the error message has been acknowledged.

- Run to Query—Start batch mode execution from the starting marker. Execution stops after the first `SQL SELECT` statement or `JISQL $DESCRIBE` or `$DUMP` macro is encountered or at the end of the script. If an error is encountered, the setting of the Continue After Error toggle determines whether or not execution is terminated.
- Single Step—Execute the current line of the script. (If the current line is blank or a comment, the next SQL statement or JISQL macro command encountered is executed.)

As execution proceeds, the script scrolls so that the current line is always in view. A bounce bar highlights the current line.

Caution: JDB does not enforce referential integrity, so an error is not returned if you insert duplicate primary keys. To prevent duplicate insertions of the same statement, you may need to move the starting marker before query execution, clearing the screen, or editing the current statement.

Once you initiate execution of the SQL script, JISQL remains in execution mode until the end of the script is encountered or until you terminate execution by choosing Reset.

To stop execution of a SQL script:

At any time, you can either:

- Choose Reset.
- Choose Run→Reset.

The Reset command stops execution of the SQL script, clears the output buffer, and resets the status of the JISQL utility so that you can edit the text of your script or restart execution.

Capturing and Displaying Query Results

When a `SQL SELECT` statement or `JISQL $DESCRIBE` or `$DUMP` macros are executed, the data retrieved can either be saved to an ASCII text file or displayed on the screen.

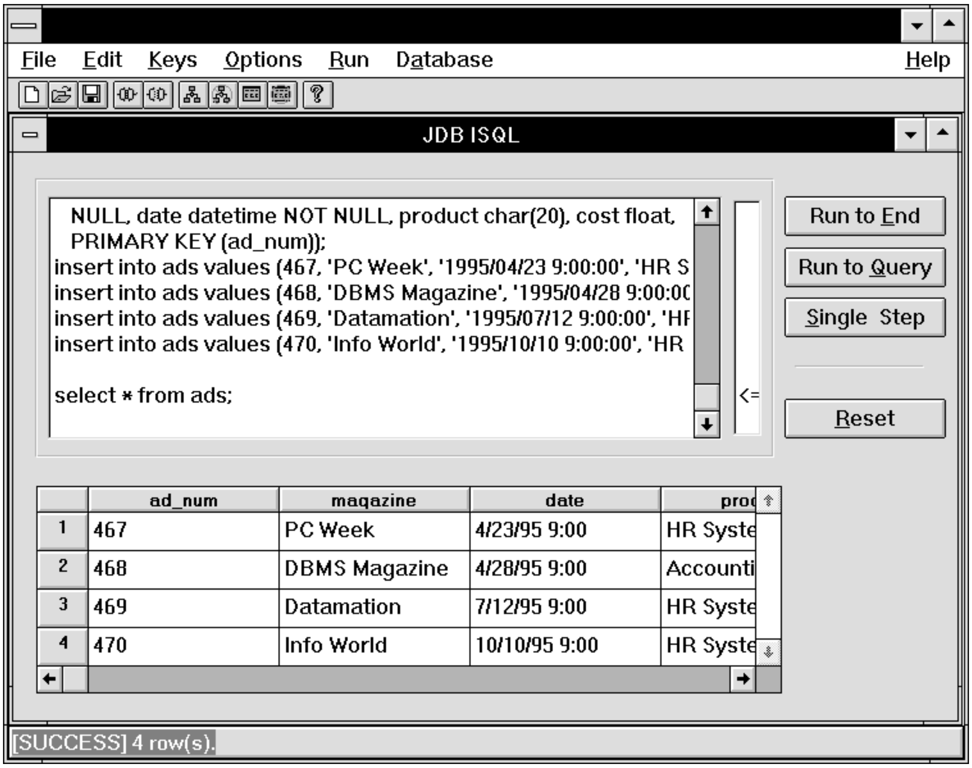
To save the output to a file:

1. Choose Options→Output to File if the Output to File toggle is not currently selected. A file selection dialog box opens.
2. Specify the name of the file for the output, and choose OK. It is recommended that you use a *.out extension in naming output files, since only files with this extension appear in the file selection dialog box.
3. Execute the script. All output generated is saved to the specified file.

Note: When output is saved to a file, select sets generated by `SQL QUERY` statements are only directed to the file and are not displayed on the screen. Output from `$DESCRIBE` and `$DUMP` macros, however, is saved in the file and displayed on the screen.

To display select sets on the screen:

1. Deselect the Output to File toggle.
2. Execute the script. When a `SQL SELECT` statement is executed, the data retrieved are displayed in the lower section of the JDB ISQL window. This area can be scrolled vertically and horizontally to view the select set.



```
  NULL, date datetime NOT NULL, product char(20), cost float,
  PRIMARY KEY (ad_num));
insert into ads values (467, 'PC Week', '1995/04/23 9:00:00', 'HR S
insert into ads values (468, 'DBMS Magazine', '1995/04/28 9:00:0(
insert into ads values (469, 'Datamation', '1995/07/12 9:00:00', 'HF
insert into ads values (470, 'Info World', '1995/10/10 9:00:00', 'HR

select * from ads;
```

	ad_num	magazine	date	proc
1	467	PC Week	4/23/95 9:00	HR Syste
2	468	DBMS Magazine	4/28/95 9:00	Accounti
3	469	Datamation	7/12/95 9:00	HR Syste
4	470	Info World	10/10/95 9:00	HR Syste

```
[SUCCESS] 4 row(s).
```

Output from `JISQL $DESCRIBE` or `$DUMP` macros is also displayed on the screen.

Creating and Viewing the Log File

You can log and then review the following information about the execution of your SQL scripts:

- Type of script execution chosen
- Text of each SQL statement or JISQL macro and the line number, as it is encountered
- Status of execution for each statement or macro, including any error messages generated

- Start and end times of script execution

How to Begin a Log Session

Choose Options⟶Record in Log. On the file selection dialog box, specify the name of the log file.

It is recommended that you use the `*.log` extension in naming log files, since only files with this extension appear in the file selection dialog box.

If you specify the name of an existing file, data from the current log session overwrites the previous contents of the file. Within a log session, however, data are appended to the file, even if you execute more than one script.

To view the information stored in the log file for the current session:

Choose Run⟶View Log File.

To end a log session:

Deselect Options⟶Record in Log.

Sample Log File

The text of log file `SESSION.LOG` follows:

```
ISQL FOR Panther, Copyright 1995-2016 Prolifics Inc.
Record Log <SESSION.LOG>: Friday April 18, 2016

<#14>:
*****  Run To End execution from line 14 of 18 at 05:54:34 *****

<#14>:   select * from ads;
         [ERROR] Table not found
********** Execution stop in line 14 of 18 at 05:54:36 **********
*****  Run To End execution from line 1 of 18 at 05:54:43 *****

<#4>:    $logon marketing;
         [SUCCESS]

<#6>:   create table ads(ad_num int NOT NULL, magazine char(20) NOT
       NULL, date datetime NOT NULL, product char(20), cost float,
         PRIMARY KEY (ad_num));
         [SUCCESS]  0 row(s).

<#9>:    insert into ads values (467, 'PC Week', '1995/04/23
9:00:00', 'HR System', 215.00);
         [SUCCESS]  1 row(s).

<#10>:   insert into ads values (468, 'DBMS Magazine', '1995/04/28
9:00:00', 'Accounting', 550.30);
         [SUCCESS]  1 row(s).

<#11>:   insert into ads values (469, 'Datamation', '1995/07/12
9:00:00', 'HR System', 312.99);
         [SUCCESS]  1 row(s).

<#12>:   select * from ads;
         [SUCCESS]  3 row(s).

<#14>:   $logon videobiz;
         [SUCCESS]

<#16>:   select * from titles where name like 'A%';
         [SUCCESS]  13 row(s).

********* Execution stop in line 19 of 18 at 05:55:03 *********
*****  New Script File  *****
***** Run To Query execution from line 1 of 6 at 05:57:22 *****

<#1>:    $logon pubs;
         [SUCCESS]

<#3>:    $describe titles;
         [SUCCESS]

********** Execution stop in line 4 of 6 at 05:57:29 **********
***** Single  Step execution from line 4 of 6 at 05:57:34 *****

<#4>:    select * from titles;
         [SUCCESS]  18 row(s).

********** Execution stop in line 5 of 6 at 05:57:40 **********
```

CHAPTER

6 SQL Reference

This chapter includes an explanation of the SQL commands and concepts in alphabetical order. You can execute the SQL commands described in this chapter using JPL procedures or using JISQL. For an example, refer to page 3-10.

Reference Organization

The reference material is listed alphabetically for the following topics:

SQL Statements

`CREATE DATABASE`

`CREATE TABLE`

`DELETE`

`DROP DATABASE`

`DROP TABLE`

`INSERT`

`SELECT`

`UPDATE`

SQL Clauses and Keywords

`BETWEEN`

`GROUP BY`

`HAVING`

`LIKE`

`ORDER BY`

`WHERE`

SQL Concepts

`Aggregate Functions`

`Data Types`

`Joins`

`Null Values`

`Operators`

`Subqueries`

Notation Conventions

This chapter includes a section for each command or topic. Each section can include the following subsections:

- Syntax
- Arguments
- Description
- Examples
- Variants
- See Also

The examples included in this section use the JISQL syntax and are based on the videobiz database. For a complete description of this database, refer to Figure 2-2 on page 2-9 or Appendix D, “Videobiz Database.”

Aggregate Functions

Obtain information about rows or groups of rows

```
functionName ([DISTINCT] expression)
```

`functionName`
: One of the following aggregate functions: `AVG`, `COUNT`, `MAX`, `MIN` or `SUM`.

`DISTINCT`
: Eliminates duplicate values before the function is applied. This keyword can be used with `AVG`, `COUNT` or `SUM`. It is not allowed with `COUNT`(*), `MAX` or `MIN`.

`expression`
: A constant, column name, subquery, or any combination of these connected by arithmetic or bitwise operators (`AND` and `OR`).

Description

Aggregate functions calculate different types of summary information on rows in a database table. All of the aggregate functions ignore null values, with the exception of `COUNT`(*). Table 6-1 lists the aggregate functions supported in JDB.

Table 6-1 Aggregate functions supported in JDB

Aggregate function	Description
`COUNT`	Counts the total number of rows retrieved with the `SELECT` statement. `COUNT(*)` calculates the number of rows retrieved. `COUNT(columnName)` calculates the number of rows containing a value in the specified column; therefore, it ignores null values.
`AVG`	Calculates and returns the average value of the specified numeric column or expression.
`MAX`	Returns the largest value of the specified column or expression.
`MIN`	Returns the lowest value of the specified column or expression.

Table 6-1 Aggregate functions supported in JDB *(Continued)*

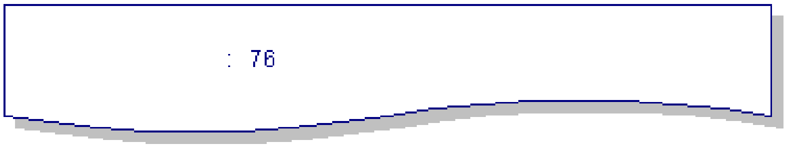
Aggregate function	Description
`SUM`	Returns the sum of the values entered in the specified numeric column or expression.

Aggregate functions generally appear in a select list, in a `HAVING` clause, or in conjunction with a `GROUP BY` clause. When used in the same statement as a `GROUP BY` clause, aggregate functions return summary information on each group of data. Aggregate functions are not valid in the `WHERE` clause of `SELECT` statements.

Example

The following statement finds the number of video titles entered in the database by querying for a count of the rows in the `titles` table:

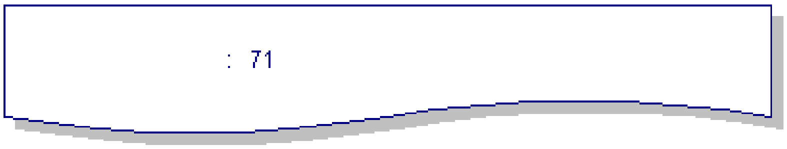
```
SELECT COUNT(*) FROM titles;
```



```
: 76
```

The following statement uses the `DISTINCT` keyword to calculate the number of video titles that have a copy of the tape available for rental.

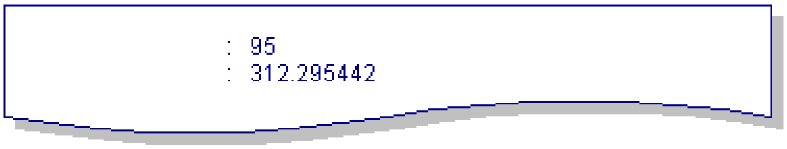
```
SELECT COUNT(distinct title_id) FROM tapes WHERE status = 'A';
```



```
: 71
```

The following statement calculates the average number of rentals per customer and the average rental amount:

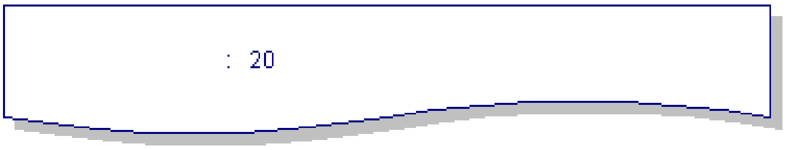
```
SELECT AVG(num_rentals), AVG(rent_amount) FROM customers;
```



```
: 95
: 312.295442
```

The following statement queries for the least number of times a copy of a video has been rented:

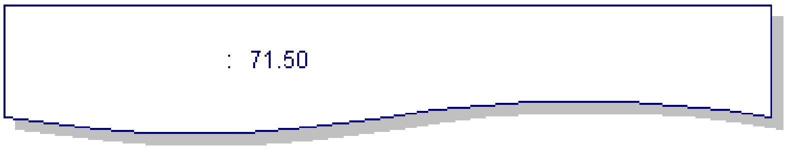
```
SELECT MIN(times_rented) FROM tapes;
```



```
:  20
```

The following statement calculates the money collected from video rentals for a particular day:

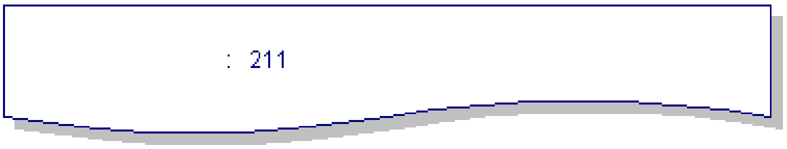
```
SELECT SUM(amount_paid) FROM rentals
   WHERE rental_date LIKE '1993/10/22%';
```



```
:  71.50
```

The following statement calculates the number of times a particular title has been rented:

```
SELECT SUM(times_rented) FROM tapes
   WHERE title_id = 12;
```



```
:  211
```

See Also GROUP BY Clause, HAVING Clause

BETWEEN Predicate

Specify a range of data values

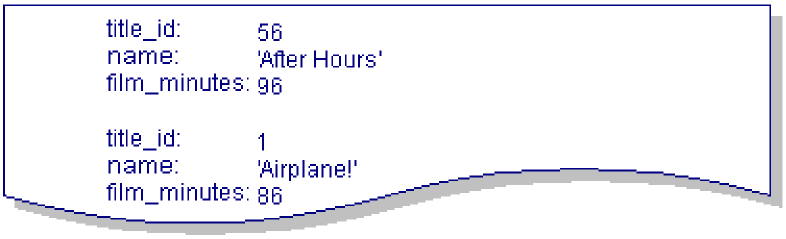
```
[NOT] BETWEEN x AND y
```

Description The `BETWEEN` predicate, located in the WHERE Clause, specifies a range of database values to be used in determining a result set. The range specified is inclusive of x and y.

If the `NOT` keyword is specified, only rows outside the specified range are included in the result set.

Example The following statement lists videos whose length is between an hour and two hours:

```
SELECT title_id, name, film_minutes FROM titles
   WHERE film_minutes BETWEEN 60 AND 120;
```



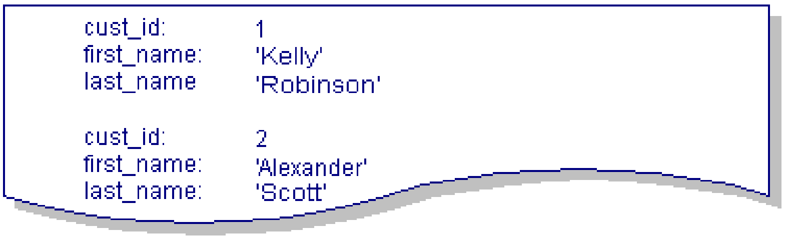
```
title_id:     56
name:         'After Hours'
film_minutes: 96

title_id:     1
name:         'Airplane!'
film_minutes: 86
```

The following statement deletes all the film rentals that occurred in 1989:

```
DELETE FROM rentals WHERE rental_date
   BETWEEN '1989/01/01 00:00:00' AND '1989/12/31 23:59:59';
```

The following statement finds which current customers live in a series of postal codes:

```
SELECT cust_id, first_name, last_name FROM customers
   WHERE postal_code BETWEEN 10200 AND 10299
   AND member_status <> 'I';
```



```
cust_id:        1
first_name:     'Kelly'
last_name       'Robinson'

cust_id:        2
first_name:     'Alexander'
last_name:      'Scott'
```

Variants The following statement performs the same query, finding the current customers in the designated series of postal codes, without the `BETWEEN` predicate:

```
SELECT cust_id, first_name, last_name FROM customers
   WHERE postal_code >= 10000 AND postal_code <= 10199
   AND member_status <> 'I';
```

See Also WHERE Clause

CREATE DATABASE Statement

Create a new database

```
CREATE DATABASE database-name
```

`database-name`
: A unique identifier for the database. Since the database appears as a file on the operating system, its identifier must follow the naming conventions for the operating system. If the database name contains characters that are not alphanumeric or if you are including a path name, the name must be enclosed in single quotation marks.

Description

The `CREATE DATABASE` statement creates a new database. A database must be created before you can declare a connection to it. You can create a database when you are connected to JDB using the identifier `@system`, when you are connected to another JDB database, or when you are using JISQL.

Creating the First Database

You can create your first database in JDB either by using JISQL or by writing a JPL procedure.

JISQL

To create the database in JISQL, first you need to start the program. For UNIX systems, it is usually located in `$SMBASE/util`. To start it, type:

```
jisql
```

Or, click on the JISQL icon.

The JDB ISQL window opens.

To create the database, choose Database⟶Create Database. The Create Database window opens.

Enter the name of the database you want to create, select the Connect after creation check box, and choose OK. This creates the database and automatically connects to it so that you can then create database tables.

JPL

The equivalent JPL procedure is as follows:

```
dbms declare syscon connection for database @system
dbms run create database database-name
dbms close connection syscon
dbms declare c1 connection for database database-name
dbms run create table table-name ...
```

Example

```
CREATE DATABASE videobiz;
```

If the database name contains non-alphanumeric characters or if you are including a path name, enclose the name in single quotation marks:

```
CREATE DATABASE 'video.db';
```

```
CREATE DATABASE '/usr/home/videobiz';
```

CREATE TABLE Statement

Creates a new database table

```
CREATE TABLE table-name (
column-name  data-type [(length)] [NOT NULL] [, column-name ...]
   [PRIMARY KEY (column-name [, column-name ... ]  ), ]
   [UNIQUE (column-name [, column-name ... ] ), ]
   [FOREIGN KEY (column-name [, column-name ... ] )
   REFERENCES table-name (column-name [, column-name ...]) [ ,] ]
   )
```

table-name
: Identifier for the table to be created. This identifier must be unique to the database. Identifiers in JDB must start with a letter but may contain letters, numbers, and underscores.

column-name
: Identifier for the column. Each column identifier must be unique within the table.

data-type
: Data type for the column. For char data types, a *length* must also be specified. For more information on data types, refer to page 6-15.

NOT NULL
: Specifies that a value must be entered for the column. The value for the column cannot be null.

PRIMARY KEY
: Specifies the primary key column(s) for this table. Any column specified as a primary key must be specified as NOT NULL.

UNIQUE
: Specifies that a column or group of columns must contain a unique entry. Any column specified as unique must be specified as NOT NULL. Column(s) specified in a PRIMARY KEY clause do not need to be declared as UNIQUE.

FOREIGN KEY
: Specifies the foreign key columns for this table. Any such column must refer to a primary or unique key in the referenced table. Matching between the foreign and primary keys is performed in the order the columns are listed, not by their names.

`REFERENCES`
: Specifies the database table and the column name in that table for the foreign key column. If more than one column is listed, the order of the columns listed in the `FOREIGN KEY` clause must match the order of the columns in the `REFERENCES` clause.

Description

The `CREATE TABLE` statement creates a new table in the current database with the specified columns. For each column, you must specify the following:

- Column name
- Data type
- Length, if the data type is char

JDB is a case-insensitive database system. No matter which case you use to enter your table and column names, JDB stores the names in lower case.

Specify Primary and Foreign Keys

You need to specify the primary and foreign keys when you create the table. The primary key is the column containing a different value in every row, which ensures that all rows are unique. In cases where one column does not perform this function, you must specify two or more columns whose values together form a unique entry. This is called a composite key. Null values are not allowed in the primary key columns; therefore, the column definitions for those columns should contain the keyword `NOT NULL`.

Foreign keys are columns in the database table that are primary or unique keys in another database table. Data entered into a foreign key column should exist as a value in the other database table. The data type for the foreign key column and its corresponding primary or unique key must be the same.

Although JDB does not enforce referential integrity based on your primary and foreign keys, it is recommended that you enter primary and foreign key information for your database tables.

Maximum Row Length

In JDB, there is a maximum row length of 1K. In other words, the sum of the table's column sizes cannot exceed 1K. The base length of the various columns is:

Data type	**Base length**
`CHAR`	Specified length
`INT`	4 bytes (stored as `LONG` in the current release)
`LONG`	4 bytes
`FLOAT`	12 bytes
`DOUBLE`	12 bytes
`DATETIME`	9 bytes

The length of a column is defined as its base length plus an additional 2 bytes for flags.

The following statement creates a table whose size equals 1028 ((255+2) * 4). Since that total is greater than 1024, JDB reports the error “Maximum record length exceeded.”

```
CREATE TABLE toobig (
    a   CHAR   (255),
    b   CHAR   (255),
    c   CHAR   (255),
    d   CHAR   (255));
```

Example

The following statement creates the actors table with `actor_id` as the primary key:

```
CREATE TABLE actors (
    actor_id            INT                NOT NULL,
    last_name           CHAR     ( 25)     NOT NULL,
    first_name          CHAR     ( 20)     ,
    PRIMARY KEY (actor_id));
```

The following statement creates the rentals table:

```
CREATE TABLE rentals (
    cust_id            INT               NOT NULL,
    title_id           INT               NOT NULL,
    copy_num           INT               NOT NULL,
    rental_date        DATETIME          NOT NULL,
    due_back           DATETIME          NOT NULL,
    return_date        DATETIME          ,
    price              FLOAT             NOT NULL,
late_fee           FLOAT            NOT NULL,
amount_paid        FLOAT            NOT NULL,
rental_status      CHAR    (  1)    NOT NULL,
rental_comment     CHAR    ( 76)    ,
modified_date      DATETIME         NOT NULL,
modified_by        CHAR    (  8)    NOT NULL,
PRIMARY KEY (cust_id, title_id, copy_num, rental_date),
FOREIGN KEY (cust_id) REFERENCES customers (cust_id),
FOREIGN KEY (title_id, copy_num)
    REFERENCES tapes (title_id, copy_num),
FOREIGN KEY (modified_by) REFERENCES users (user_id));
```

See Also   Data Types

Data Types

List the data types available in JDB

Description The JDB data types are described in this section.

CHAR (n)

Character column containing ASCII characters (letters, numbers and symbols). Specify the maximum size of the column with n. n can range in value from 1 to 255. The size of a `CHAR` column is n no matter how many characters are entered into the column. If the character string is longer than n, the string is truncated to the specified length. If the character string is shorter than n, the string is blank-padded to the specified length. For example, an entry of
'A12'
in a `CHAR`(4) column would be stored as
'A12 '

The storage size of a `CHAR` column is n plus 2 bytes for flags.

To enter values into `CHAR` columns using JISQL, enclose the character string in single quotation marks. To include a single quotation mark as part of the entry, enter two consecutive single quotation marks.

If you use colon plus processing or binding in a JPL procedure, Panther automatically formats the character string by enclosing the character string in single quotation marks and converting each single quotation mark to two single quotation marks.

INT

Numeric column containing whole numbers. In the current version of JDB, all `INT` values are stored as `LONG` values.

LONG

Numeric column containing whole numbers ranging from -2,147,483,647 to +2,147,483,647. The storage size for an `LONG` column is 4 bytes plus 2 bytes for flags.

FLOAT

Numeric column containing positive or negative floating point numbers. The hardware platform determines the precision and range of `FLOAT` columns. The storage size is 12 bytes plus 2 bytes for flags.

DOUBLE

Numeric column containing double precision numbers. The hardware platform determines the precision and range of `DOUBLE` columns. The storage size is 12 bytes plus 2 bytes for flags.

DATETIME

Date column containing both a date and time of day. The storage size is 9 bytes, plus 2 bytes for flags. The default format for a `DATETIME` column is: yyyy/mm/dd hh:mm:ss

For example, January 28, 1993 at 2:40 p.m. is entered as follows:

1993/01/28 14:40:00

Alternate formats for `DATETIME` values include using periods instead of colons to separate time entries and using spaces instead of slashes to separate date entries.

To enter `DATETIME` values in JISQL, enclose the date entry in single quotes as in:

'1993/01/28 14:40:00'

To enter `DATETIME` values in JPL, the date should both be enclosed in single quotes and contain double colons:

```
DBMS RUN UPDATE titles \
SET release_date = '1994/01/28 00::00::00' \
WHERE title_id = :+title_id
```

If `DT_DATETIME` is the Panther type of the widget containing the entry, Panther automatically formats the date according to the specified Date/Time format.

The data type of each column is stored in the system table, `syscols`. You can query this table to find the data type for any column. Refer to page 4-3 for more information on the `syscols` table.

Numeric Columns

In JDB, you cannot enter numbers with leading zeros in numeric columns.

Example

The following statement creates the titles table:

```
CREATE TABLE titles (
    title_id          INT              NOT NULL,
    name              CHAR    ( 60)    NOT NULL,
    genre_code         CHAR    (  4)   ,
    dir_last_name      CHAR    ( 25)   ,
    dir_first_name     CHAR    ( 20)   ,
    film_minutes       INT             ,
    rating_code        CHAR    (  4)   ,
    release_date       DATETIME        ,
    pricecat           CHAR    (  1)   NOT NULL
    PRIMARY KEY (title_id),
    FOREIGN KEY (pricecat) REFERENCES pricecats (pricecat));
```

The `titles` table contains columns of various data types. The following statement inserts a row into this table:

```
INSERT INTO titles (title_id, name, genre_code,
   dir_last_name, dir_first_name, film_minutes, rating_code,
   release_date, pricecat)
   VALUES (72, 'Howards End', 'DRAM', 'Ivory', 'James', 140,
   'PG', '1992/01/01 00:00:00', 'G');
```

DELETE Statement

Remove information from a database table

```
DELETE FROM table-name [WHERE search-conditions]
```

`table-name`
Identifier for the database table.

`WHERE`
The `WHERE` clause specifies which rows will be deleted. Refer to page 6-54 for more information on the `WHERE` clause.

Description The `DELETE` statement removes a row or rows from the specified table. To keep data consistent across a database, you may need to delete or update rows in other database tables whose values depend on the deleted row.

Warning: If no `WHERE` clause is specified, all the information in the table is deleted.

Example If a customer drops his membership, you can delete that customer from the database:

```
DELETE FROM customers WHERE cust_id = 123;
```

To delete a video title from the database, you would need to delete rows from titles, `title_dscr`, tapes and roles:

```
DELETE FROM title_dscr WHERE title_id = 134;
DELETE FROM roles WHERE title_id = 134;
DELETE FROM tapes WHERE title_id = 134;
DELETE FROM titles WHERE title_id = 134;
```

You can delete rows using a subquery in the `WHERE` clause:

```
DELETE FROM actors WHERE actor_id IN
   (SELECT actor_id FROM roles WHERE title_id = 134);
```

See Also `WHERE Clause`

DROP DATABASE Statement

Remove a database

```
DROP DATABASE database-name
```

`database-name`
Name of the database to be removed.

Description

The `DROP DATABASE` statement deletes the specified database. The file containing the database is removed from the operating system. If the database name contains characters that are not alphanumeric, enclose the name in single quotation marks.

You cannot drop the current database. First, you must close the connection with the current database and connect either to another database or to the system catalog.

When you drop a database, the journal files are not deleted.

Example

```
DROP DATABASE videobiz;
```

Enclose the name in single quotation marks if it contains non-alphanumeric characters.

```
DROP DATABASE 'video.db';
```

DROP TABLE Statement

Remove a table from the database

```
DROP TABLE table-name
```

`table-name`
Name of the table to be deleted.

Description The `DROP TABLE` statement deletes the specified table from the database, including the data stored in the table.

Example

```
DROP TABLE rentals;
```

GROUP BY Clause

Divide the returned data into groups according to the specified column(s)

```
GROUP BY [correlation-name.column-name[ , ... ]
```

correlation-name
: Identifier which substitutes for the table name.

column-name
: Column used to group the data.

Description

A `GROUP BY` clause included in a `SELECT` statement lets you specify the column or columns to be used to divide the table into groups. Rows having an identical value in the specified columns are grouped together.

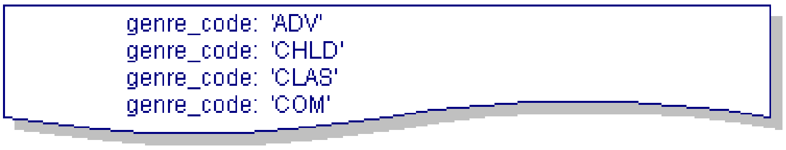
A `GROUP BY` clause is most often combined with an aggregate function in order to obtain summary information on each group. A `GROUP BY` clause can also be followed by a `HAVING` clause in order to define which groups appear in the result set.

In a `SELECT` statement containing a `GROUP BY` clause, the columns specified in the select list or in the `HAVING` clause must either be listed in the `GROUP BY` clause or be parameters of aggregate functions.

Example

This statement finds the number of videos in each rating category:

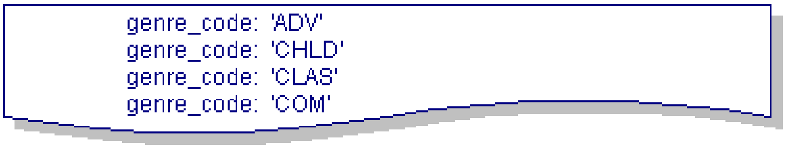
```
SELECT rating_code, COUNT(*) FROM titles
    GROUP BY rating_code;
```



```
genre_code: 'ADV'
genre_code: 'CHLD'
genre_code: 'CLAS'
genre_code: 'COM'
```

A `GROUP BY` clause can be used to find unique entries in a `SELECT` statement; however, the `DISTINCT` generally used for this purpose. The following statement lists the types of videos found in the titles table:

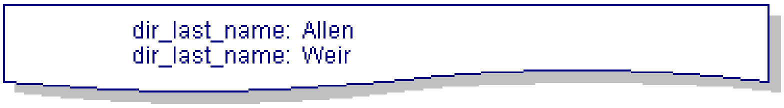
```
SELECT genre_code FROM titles GROUP BY genre_code;
```



```
genre_code: 'ADV'
genre_code: 'CHLD'
genre_code: 'CLAS'
genre_code: 'COM'
```

This statement using both a `GROUP BY` clause and a `HAVING` clause determines the people who directed more than three videos:

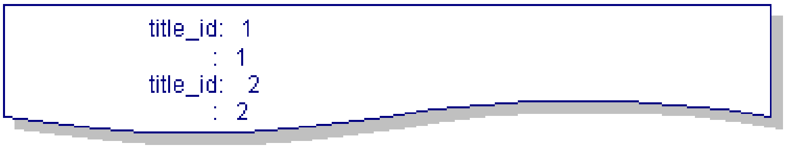
```
SELECT dir_last_name FROM titles GROUP BY dir_last_name
   HAVING COUNT(*) > 3;
```



```
dir_last_name: Allen
dir_last_name: Weir
```

If your `SELECT` statement also includes a `WHERE` clause, place the `GROUP BY` clause after `WHERE` clause.

```
SELECT title_id, COUNT (*) FROM tapes WHERE status = 'A'
   GROUP BY title_id;
```



```
title_id:  1
       :  1
title_id:  2
       :  2
```

See Also Aggregate Functions, HAVING Clause, SELECT Statement

HAVING Clause

Set search conditions in order to obtain a subset of data

```
HAVING search-conditions
```

search-conditions
Specifies the conditions for the selection of data. For a complete listing of available conditions, refer to page 6-54.

Description

A `HAVING` clause included in a `SELECT` statement allows you to select a subset of data which has a certain value.

Generally, a `HAVING` clause appears in conjunction with a `GROUP BY` clause. When this occurs, the `HAVING` clause selects its subsets after the `GROUP BY` clause has been applied.

Unlike the `WHERE` clause, a `HAVING` clause can include aggregate functions.

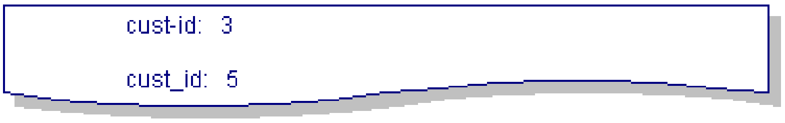
In statements using both a `WHERE` clause and a `HAVING` clause, the following steps occur:

1. The `WHERE` clause selects the rows meeting its search conditions.
2. The `GROUP BY` clause divides these rows into groups according to the specified column(s).
3. The `HAVING` clause excludes groups not meeting its search conditions.
4. Any aggregate function specified in the select list performs its calculations for each group.

Example

The following statement finds the customers that are frequent renters for the month:

```
SELECT cust_id FROM rentals
   WHERE rental_date
   BETWEEN '1993/10/01 00:00:00' AND '1993/10/31 23:59:59'
   GROUP BY cust_id
   HAVING COUNT (*) > 4;
```



```
cust-id:  3

cust_id:  5
```

See Also   Aggregate Functions, GROUP BY Clause, SELECT Statement, WHERE Clause

INSERT Statement

Add information to a database table

```
INSERT INTO table-name [(column-list)] VALUES (literal | NULL [ ,
... ])
INSERT INTO table-name [(column-list)] query-expression
```

table-name
: Unique identifier for the database table.

column-list
: Columns which will have values inserted. See the description below.

VALUES
: Columns which will have values inserted. See the description below.

query-expression
: Subquery used to specify data to be inserted.

Description

The INSERT statement enters information into the specified table. There are two forms of the INSERT statement. In the first form, you insert a single row by specifying values for the specified columns. In the second form, you use a query to select rows from other tables to be inserted into the specified table.

Within the first form of the INSERT statement, several format variations exist. The simplest format includes a VALUES clause without a column list. In this format, you must provide a value for each column in the table. The values are listed in the same order that was used to create the columns in the database table.

```
INSERT INTO roles
   VALUES (72, 144, 'Margaret Schlegel');
```

In a VALUES clause, the column values are separated by commas. You can enter character strings, date strings, and numeric constants as column values. If you are entering the data using JISQL, character strings and date values must be enclosed with single quotation marks.

Inserting Rows Using a Column List

If you do not know the column order or if you do not want to enter a value for each column, you can add a column list to the statement:

```
INSERT INTO roles (title_id, actor_id, role)
   VALUES (72, 144, 'Margaret Schlegel');
```

With this format, the first column value, 72, is entered into the first column found in the column list, `title_id`. The second value goes into the second column listed, etc.

If you do not specify a value for a column, its value will be set to `NULL`.

Inserting Rows Containing a Null Value

You can also enter an unknown value for any column using `NULL` as the column value:

```
INSERT INTO roles (title_id, actor_id, role)
   VALUES (72, 144, NULL);
```

However, this syntax is not available if the column was specified as `NOT NULL` in the CREATE TABLE Statement.

Inserting Rows Using a Subquery

The second syntax statement illustrates the insertion of rows using a subquery. Multiple rows can be inserted with this format; however, you cannot have the same table named in the `INTO` clause and the `SELECT` statement of the query.

```
INSERT INTO roles
   (title_id)
   SELECT title_id FROM titles WHERE title_id > 75;
```

See Also   Null Values

Joins

Specify the interconnection between two tables

```
... FROM table-name, table-name
   WHERE table-name.column-name join-operator
      table-name.column-name
   [ {AND|OR|NOT} table-name. column-name join-operator
      table-name.column-name ... ]
```

`FROM`
: The `FROM` clause lists the tables included in the join.

`table-name`
: Identifier for the database table.

`WHERE`
: The `WHERE` clause specifies the relationship between each set of tables in addition to the search conditions to be used for the statement.

`column-name`
: Column from one of the specified database tables.

`join-operator`
: One of the following operators: =, >, <, >=, <=, or <>.

Description

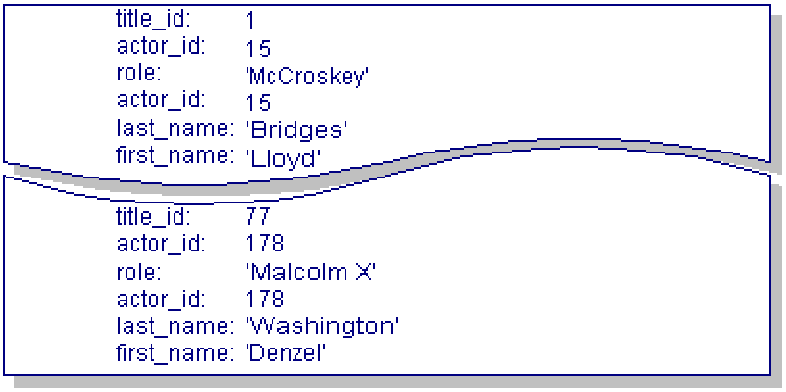
A join connects two or more database tables by specifying the relationship between each set of tables. To specify the relationship, you connect one column from one table to a column in another table. The column names must be qualified with the table name if the table location is ambiguous. A join can be part of a `SELECT`, `UPDATE`, `INSERT`, or `DELETE` statement. A join can also be included in a subquery. There are several types of joins which will be discussed in the following paragraphs.

Equi-joins

An equi-join is based on equality as indicated by the equal sign (=). In an `equi-join`, all the columns in the tables being joined are included in the result set. For example,

```
SELECT * FROM roles, actors
   WHERE roles.actor_id = actors.actor_id;
```

This statement joins the actors and roles tables using the `actor_id` column in each table. The result set lists the actor for each role included in the roles table.



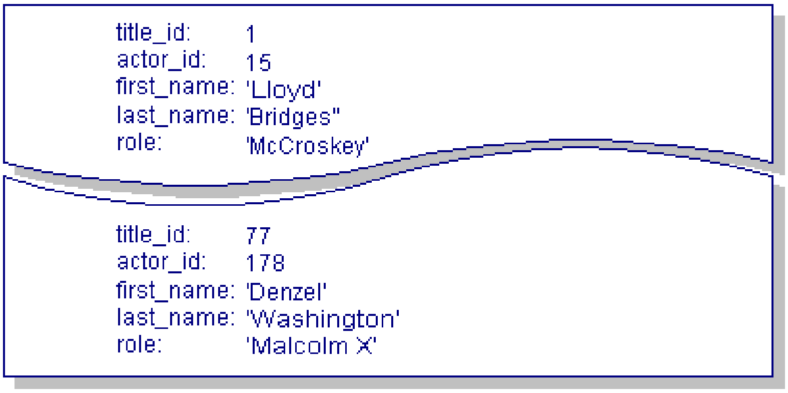
```
title_id:   1
actor_id:   15
role:       'McCroskey'
actor_id:   15
last_name:  'Bridges'
first_name: 'Lloyd'

title_id:   77
actor_id:   178
role:       'Malcolm X'
actor_id:   178
last_name:  'Washington'
first_name: 'Denzel'
```

Natural Joins

A natural join is structured so that there is no duplication of data. The same query as a natural join would appear as follows:

```
SELECT title_id, roles.actor_id, first_name, last_name, role
    FROM roles, actors
    WHERE roles.actor_id = actors.actor_id;
```

The select list names the columns to be included so that the `actor_id` is displayed only once.

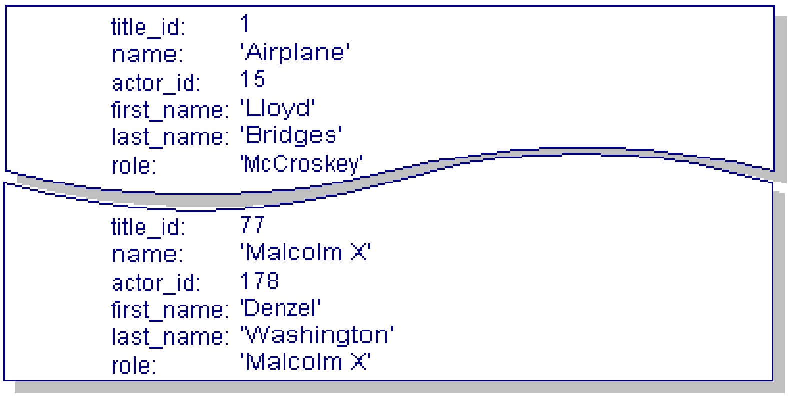


```
title_id:   1
actor_id:   15
first_name: 'Lloyd'
last_name:  'Bridges"
role:       'McCroskey'

title_id:   77
actor_id:   178
first_name: 'Denzel'
last_name:  'Washington'
role:       'Malcolm X'
```

Multiple Table Joins

A multiple table join involves more than two tables using one or more columns to make the connection. The following statement adds the name of the video to the result set.

```
SELECT roles.title_id, titles.name, actors.actor_id,
   actors.first_name, actors.last_name, roles.role
   FROM roles, titles, actors
   WHERE roles.title_id = titles.title_id
   AND roles.actor_id = actors.actor_id;
```



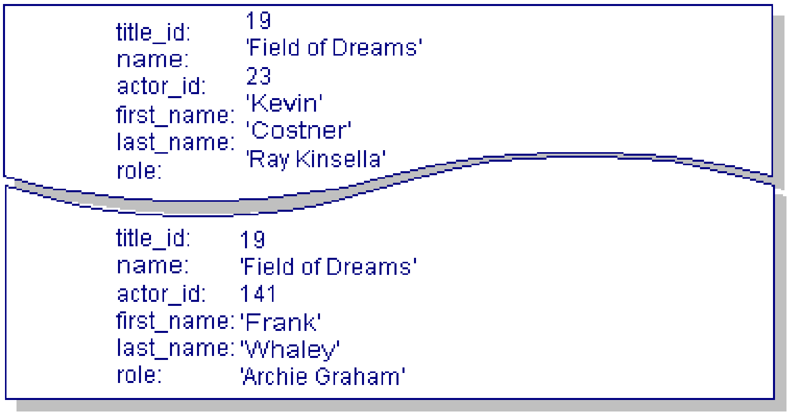
```
title_id:   1
name:       'Airplane'
actor_id:   15
first_name: 'Lloyd'
last_name:  'Bridges'
role:       'McCroskey'

title_id:   77
name:       'Malcolm X'
actor_id:   178
first_name: 'Denzel'
last_name:  'Washington'
role:       'Malcolm X'
```

You could also use correlation names to formulate the query:

```
SELECT r.title_id, t.name, a.actor_id,
   a.first_name, a.last_name, r.role
   FROM roles r, titles t, actors a
   WHERE r.title_id = t.title_id
   AND r.actor_id = a.actor_id;
```

Additional search conditions can be added to the `WHERE` clause to further restrict the result set:

```
SELECT roles.title_id, titles.name, actors.actor_id,
   actors.first_name, actors.last_name, roles.role
   FROM roles, titles, actors
   WHERE roles.title_id = titles.title_id
   AND roles.actor_id = actors.actor_id
   AND titles.title_id = 19;
```



title_id: 19
name: 'Field of Dreams'
actor_id: 23
first_name: 'Kevin'
last_name: 'Costner'
role: 'Ray Kinsella'

title_id: 19
name: 'Field of Dreams'
actor_id: 141
first_name: 'Frank'
last_name: 'Whaley'
role: 'Archie Graham'

Non equi-joins

In addition to the equal sign, there are additional operators that can be specified. Table 6-2 lists all the relational operators that can be used in joins.

Table 6-2 Join operators

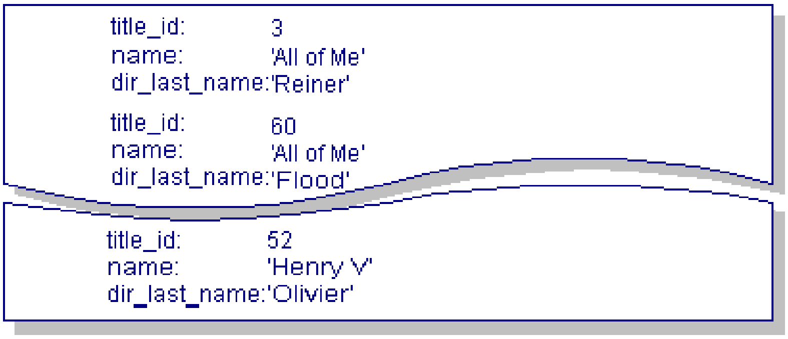
Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

You can also use a `BETWEEN` predicate to specify a range of values.

The following query lists the videos that have the same name but have been directed by different people:

```
SELECT t.title_id, t.name, t.dir_last_name
   FROM titles t, titles d
```

```
    WHERE t.name = d.name
    AND t.dir_last_name <> d.dir_last_name;
```



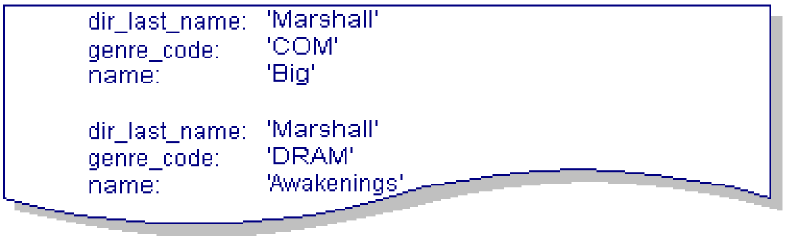
```
title_id:       3
name:           'All of Me'
dir_last_name:'Reiner'

title_id:       60
name:           'All of Me'
dir_last_name:'Flood'

title_id:       52
name:           'Henry V'
dir_last_name:'Olivier'
```

Self-joins

The previous query is called a self-join which joins a table to itself so that you can compare values in the same column. To make a self-join, use correlation names for the database tables in the `FROM` clause and in the column names.

The following self-join finds the directors who have made two different types of films–for example, directors who have made both comedy and adventure films. All of this information is in the titles table. For this query, the join condition is made on the director's last name. Then, the two `genre_code` entries in each join are compared, and if they differ, the director's last name, the genre code and the name of each corresponding video are written to the result set.

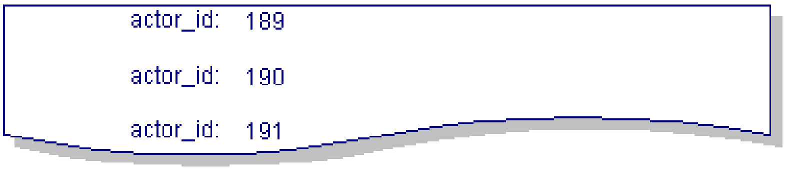
```
SELECT dir.dir_last_name, dir.genre_code, dir.name
    FROM titles gen, titles dir
    WHERE gen.dir_last_name = dir.dir_last_name
    AND gen.genre_code <> dir.genre_code
```



```
dir_last_name:  'Marshall'
genre_code:     'COM'
name:           'Big'

dir_last_name:  'Marshall'
genre_code:     'DRAM'
name:           'Awakenings'
```

The following self-join finds the actors in video #50 who are entered in the roles table only for that video. It uses one version of the roles table to find all the `actor_id` codes in video #50. It uses the other version of the table to find the `actor_id` codes that are entered in the roles table only once.

```
SELECT r.actor_id FROM roles r, roles j
   WHERE r.title_id = 50
   AND r.actor_id = j.actor_id
   GROUP BY r.actor_id HAVING COUNT(j.actor_id) = 1;
```



```
actor_id:   189

actor_id:   190

actor_id:   191
```

See Also Subqueries

LIKE Predicate

Obtain data matching a specified pattern

```
column-name [NOT] LIKE literal [ESCAPE literal]
```

`column-name`
: Column whose value you want to specify.

`literal`
: Wildcard characters intermixed with portions of column values.

Description

A `LIKE` predicate selects rows in which a column value matches a specified pat tern. You can enter values for character strings or date strings. You can also enter a wildcard character to substitute for a portion of the string. Table 6-3 lists the wild card characters that can be used in JDB.

Table 6-3 Wildcard characters

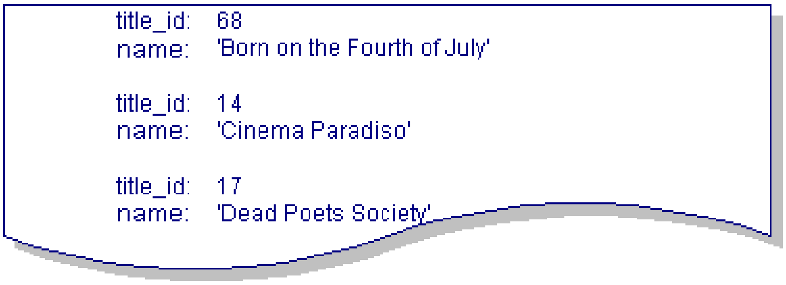
Wildcard	Description
% (percent sign)	Substitutes for any string of zero or more characters
_ (underline)	Substitutes for any single character

With the specification of an `ESCAPE` clause, the special meaning given to “_” and “%” can be disabled. `NOT LIKE` selects rows that do not match the specified pattern.

Example

This query finds all the videos released in 1989:

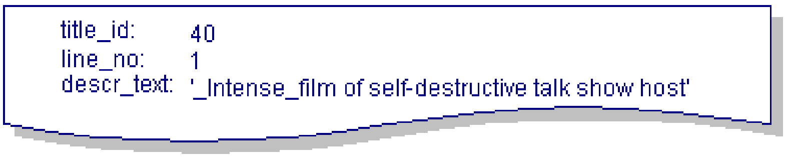
```
SELECT title_id, name FROM titles
   WHERE release_date LIKE '1989%';
```



```
title_id:   68
name:       'Born on the Fourth of July'

title_id:   14
name:       'Cinema Paradiso'

title_id:   17
name:       'Dead Poets Society'
```

The following example returns rows where the `dscr_text` begins with an underscore. The backslash removes the special meaning for the underscore, but not for the percent sign:

```
SELECT * FROM title_dscr
   WHERE dscr_text LIKE '\_%' ESCAPE ‘\';
```



```
title_id:    40
line_no:     1
descr_text:  '_Intense_film of self-destructive talk show host'
```

See Also WHERE Clause

Null Values

Specify an unknown value

In `INSERT` statements

```
... VALUES {literal | NULL} [ , {literal | NULL} ]
```

In `SELECT` statements,

```
... WHERE column-name IS [NOT] NULL
```

In `UPDATE` statements

```
... SET column-name  = {literal | NULL}
... [ , column-name = {literal | NULL} ]
```

Description

When a column is set to `NULL`, it specifies an unknown or an unspecified value. A `NULL` value is not the same as a blank or a zero entered into a column.

If you are using a comparison operator, be aware that `NULL` is not a value and therefore cannot be compared to any other value. As an example, the following `WHERE` clause would evaluate to true for all values of the `times_rented` column that are greater than 75, but would evaluate to false if the column is set to `NULL`.

```
WHERE times_rented > 75
```

Examples

The examples illustrate the uses of `NULL` values in different types of statements.

The following statement inserts a null value into the role column:

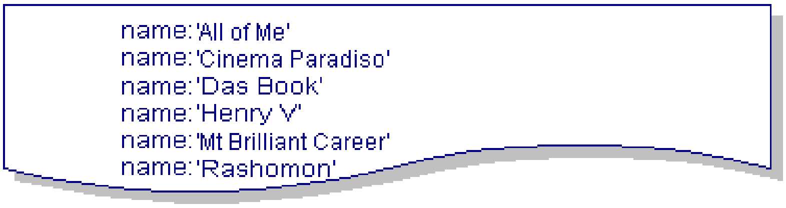
```
INSERT INTO roles (title_id, actor_id, role)
   VALUES (16, 276, NULL);
```

An error occurs if you attempt to insert a null value into a column which was created as `NOT NULL`. The following statement returns the error `NULL` not allowed since the column `actor_id` was specified as `NOT NULL` in the `CREATE TABLE` statement for the roles table.

```
INSERT INTO roles (title_id, actor_id, role)
   VALUES (27, NULL, 'Aunt Gussie');
```

The following statement selects rows where the `rating_code` column contains a null value:

```
SELECT name FROM titles WHERE rating_code IS NULL;
```



```
name:'All of Me'
name:'Cinema Paradiso'
name:'Das Book'
name:'Henry V'
name:'Mt Brilliant Career'
name:'Rashomon'
```

The following statement updates the `rental_comment` column to a null value for every row in the rentals table:

```
UPDATE rentals SET rental_comment = NULL;
```

The following statement updates the `rental_comment` to a null value for a specific rental:

```
UPDATE rentals SET rental_comment = NULL
   WHERE cust_id = 6
   AND title_id = 69
   AND copy_num = 2
   AND rental_date = '1993/10/29 18:00:00';
```

In order to obtain a unique entry for the rentals table, you must include an entry for the `cust_id`, `title_id`, `copy_num` and `rental_date` columns.

Operators

Description This section describes the various operators available in JDB.

Arithmetic Operators

Arithmetic operators allow you to perform calculations on data in the database without altering the data. They are available to use with any numeric column. If the value in the column is `NULL`, the result will also be `NULL`.

Table 6-4 lists the arithmetic operators that are available in JDB.

Table 6-4 Arithmetic operators

Operator	Definition
+	Addition
-	Subtraction
*	Multiplication
/	Division

The arithmetic operators adhere to the following order of precedence:

1. multiplication, division
2. subtraction, addition

Among operators that have the same level of precedence, the order of execution is from left to right. The order of precedence can also be explicitly specified using parentheses. For more information, refer to the section on logical operators.

The following statement uses an arithmetic operator to calculate the price with sales tax on an item:

```
SELECT pricecat, price * 1.0825 FROM pricecats
   WHERE pricecat = 'N';
```

Arithmetic operators can also be used in calculations that perform comparisons. The following statement finds rentals where the amount paid was greater than double the rental fee:

```
SELECT cust_id, title_id, rental_date FROM rentals
   WHERE amount_paid > price * 2;
```

Comparison Operators

Comparison operators allow you to compare one expression with another expression, where an expression is defined as a column name, a constant, a function, or any combination of column names, constants and functions.

Table 6-5 lists the comparison operators that are available in JDB.

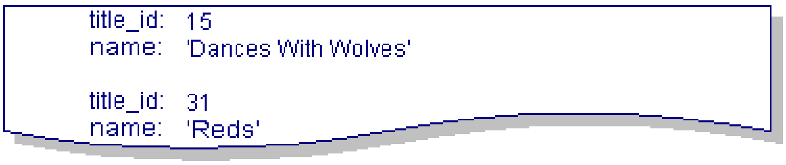
Table 6-5 Comparison operators

Operator	Definition
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

When character or date strings are used in comparisons, they need to be enclosed in single quotation marks. Also, in these comparisons, numbers are greater than uppercase letters, and uppercase letters are greater than lowercase letters. For character strings, > asks for character strings closer to the end of the alphabet, < for character strings closer to the beginning of the alphabet. For date strings, > asks for dates later than the one specified and < asks for dates earlier than the one specified.

The following query asks for the videos whose length is greater than three hours:

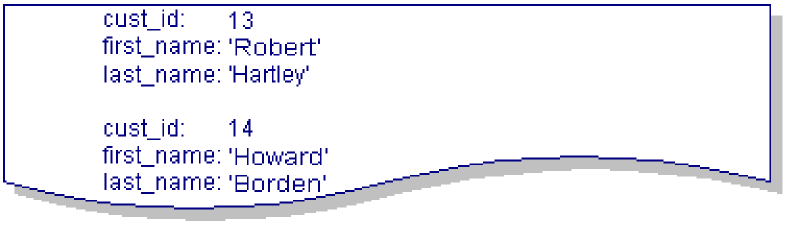
```
SELECT title_id, name FROM titles WHERE film_minutes > 180
```



```
title_id:  15
name:      'Dances With Wolves'

title_id:  31
name:      'Reds'
```

The following query lists the customers who joined during the current year:

```
SELECT cust_id, first_name, last_name FROM customers
   WHERE member_date >= '1993/01/01 00:00:00';
```

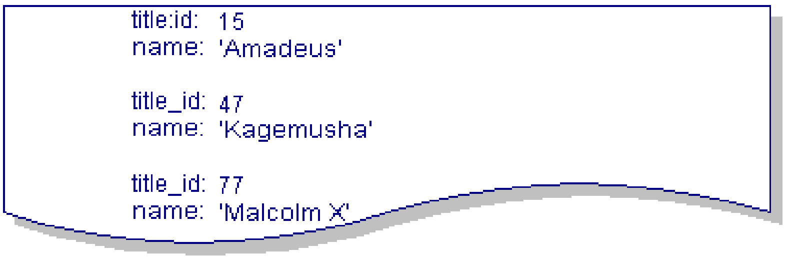


```
cust_id:     13
first_name: 'Robert'
last_name: 'Hartley'

cust_id:     14
first_name: 'Howard'
last_name: 'Borden'
```

Querying for a specific range of values can be accomplished using a series of comparison operators or a BETWEEN predicate. The following statements would return the same data:

```
SELECT title_id, name FROM titles
   WHERE film_minutes BETWEEN 150 AND 180;
```

```
SELECT title_id, name FROM titles
   WHERE film_minutes >=150 AND film_minutes <= 180;
```



```
title:id:  15
name:  'Amadeus'

title_id:  47
name:  'Kagemusha'

title_id:  77
name:  'Malcolm X'
```

Logical Operators

Logical operators join sets of search conditions together.

Table 6-6 lists the logical operators that are available in JDB.

Table 6-6 Logical operators

Operator	Definition

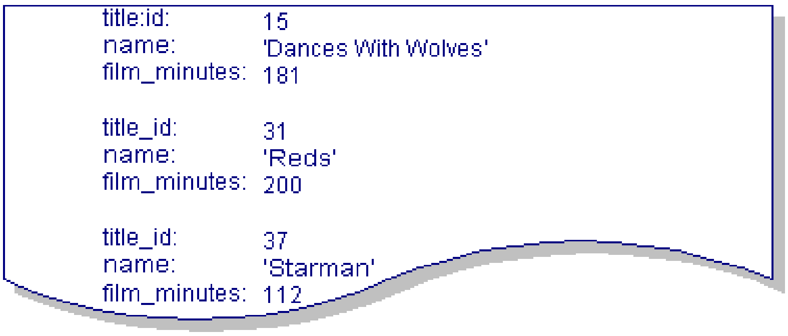
Table 6-6 Logical operators

AND	Joins two conditions and returns results when both conditions are true.
OR	Joins two conditions and returns results when either condition is true.

AND operators take precedence over OR operators unless you change the order of execution by using parentheses. Also, NOT takes precedence over AND.

If you wanted to find the science fiction videos that either have a PG or G rating or that are over three hours long, the following query would not return the correct results. This query first finds the science fiction videos that have a PG or G rating. Then, it finds any videos over three hours long.

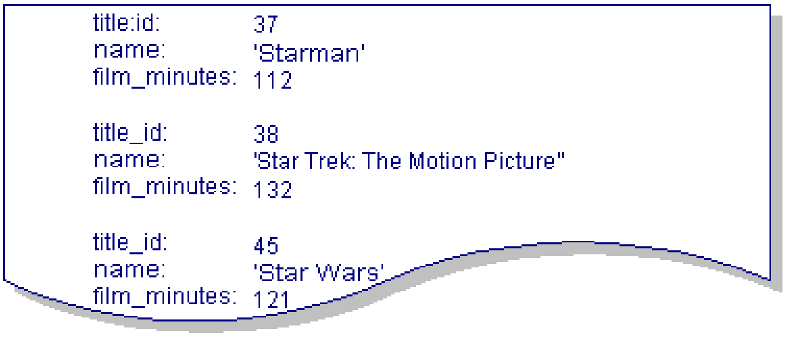
```
SELECT title_id, name, film_minutes FROM titles
   WHERE genre_code = 'SCFI'
   AND rating_code IN ('G', 'PG')
   OR film_minutes > 180;
```



```
title:id:      15
name:          'Dances With Wolves'
film_minutes:  181

title_id:      31
name:          'Reds'
film_minutes:  200

title_id:      37
name:          'Starman'
film_minutes:  112
```

The addition of parentheses finds science fiction videos that either have a PG or G rating or that are over three hours long.

```
SELECT title_id, name, film_minutes FROM titles
   WHERE genre_code = 'SCFI'
   AND (rating_code IN ('G', 'PG')
   OR film_minutes > 180);
```



```
title:id:      37
name:          'Starman'
film_minutes:  112

title_id:      38
name:          'Star Trek: The Motion Picture"
film_minutes:  132

title_id:      45
name:          'Star Wars'
film_minutes:  121
```

Remember that if a column is set to `NULL`, no comparison operator will retrieve it. The value of null is unknown. The following example would find the video titles whose length is less than an hour but would not find the ones whose length is entered as `NULL`:

```
SELECT title_id, name FROM titles
   WHERE film_minutes < 60;
```

ORDER BY Clause

Specify the order for the query results

```
ORDER BY {integer | [correlation-name.]column-name} [ , ... ]
   [ASC | DESC]
```

integer
: If integer is specified instead of a column-name, it refers to the position of a column or expression in the select list.

correlation-name
: Identifier which substitutes for the table name.

column-name
: Specifies the column or columns to be used for sorting the result set.

ASC
: Specifies that the result set is to be sorted in ascending order. This is the default.

DESC
: Specifies that the result set is to be sorted in descending order.

Description

An ORDER BY clause sorts the result set according to the specified column or columns. The columns specified in the ORDER BY clause must also be specified in the select list of the SELECT statement. By default, the sort occurs in ascending order which lists the smallest value first. You can set the sort order by specifying ASC for ascending or DESC for descending order.

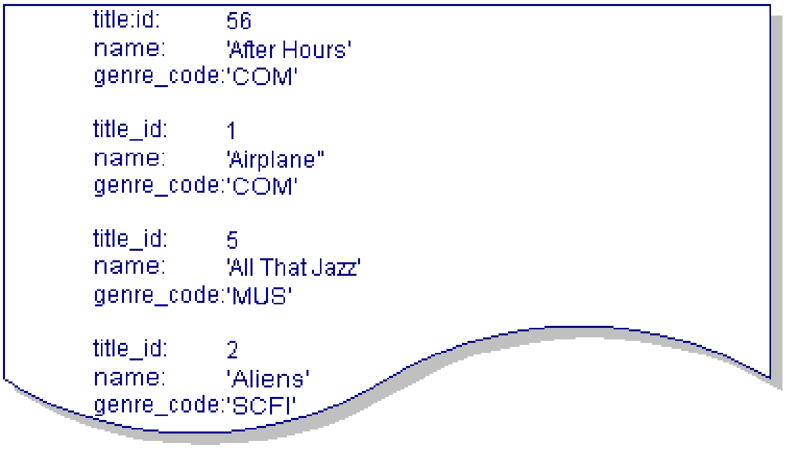
If you list more than one column in the ORDER BY clause, it creates a nested sort. The sort for the first column takes precedence and occurs first. Then, within each of these groups, the rows are sorted again according to the value of the second column.

Instead of listing column names in the ORDER BY clause, you can use integers to refer to the column position.

Example

The following SELECT statement without an ORDER BY clause returns the list of video titles in the order shown in the result set:

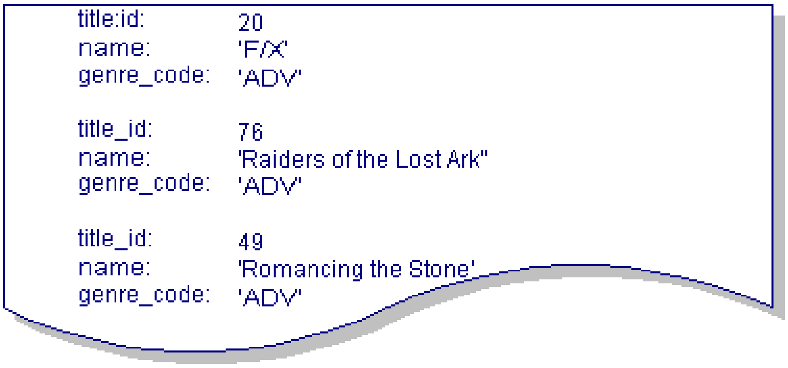
```
SELECT title_id, name, genre_code FROM titles;
```



```
title:id:     56
name:         'After Hours'
genre_code:'COM'

title_id:     1
name:         'Airplane"
genre_code:'COM'

title_id:     5
name:         'All That Jazz'
genre_code:'MUS'

title_id:     2
name:         'Aliens'
genre_code:'SCFI'
```

With the addition of an `ORDER BY` clause on the genre code, followed by the video name, the statement returns the data in the following order:

```
SELECT title_id, name, genre_code FROM titles
    ORDER BY genre_code, name;
```



```
title:id:          20
name:              'F/X'
genre_code:        'ADV'

title_id:          76
name:              'Raiders of the Lost Ark"
genre_code:        'ADV'

title_id:          49
name:              'Romancing the Stone'
genre_code:        'ADV'
```

If you use integers in the `ORDER BY` clause to refer to the column position, the previous statement appears in the following syntax:

```
SELECT title_id, name, genre_code FROM titles
    ORDER BY 3, 2;
```

`genre_code` is the third column appearing in the select list, and name is the second column in the select list.

SELECT Statement

Obtain information from a database table

```
SELECT [DISTINCT] {select-list | *} FROM table-name
       [correlation-name] [, ...]
    [WHERE search-conditions]
    [GROUP BY [correlation-name.]column-name[, ...]]
    [HAVING search-conditions]
    [ORDER BY {integer | [correlation-name.]column-name} [, ...]]
```

`DISTINCT`
: Exclude any duplicate rows from the result set.

`select-list`
: A series of column names, qualified by the table name if more than one database table is being accessed, and/or aggregate functions.

`*`
: Selects every column from every table listed in the `FROM` clause.

`table-name`
: Identifier for a database table.

`correlation-name`
: Identifier which substitutes for the table name in the remainder of the statement.

`WHERE`
: The `WHERE` clause specifies a search condition or a join. For more information on joins, refer to page 6-27. For more information on the `WHERE` clause, refer to page 6-54.

`search-conditions`
: Specifies the conditions for the selection of data. For more information, refer to page 6-54.

`GROUP BY`
: The `GROUP BY` clause specifies the column used to divide the result set into groups. For more information, refer to page 6-21.

`HAVING`
: The `HAVING` clause specifies a search condition. For more information, refer to page 6-23

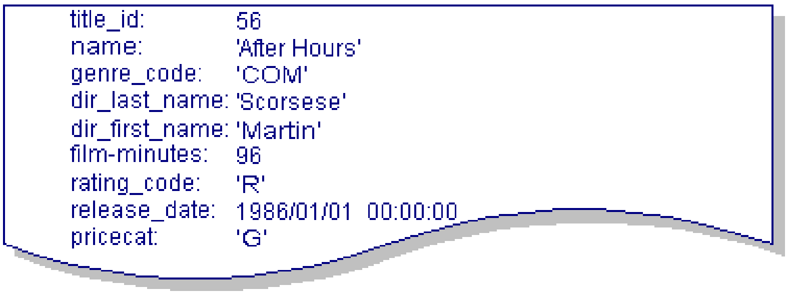
`ORDER BY`

The `ORDER BY` clause specifies the column(s) used to sort the result set. For more information, refer to page 6-42.

Description

The SELECT statement obtains data from the specified database table or tables. In its simplest form, the SELECT statement retrieves all the data from all the columns in the named table:

```
SELECT * FROM titles;
```



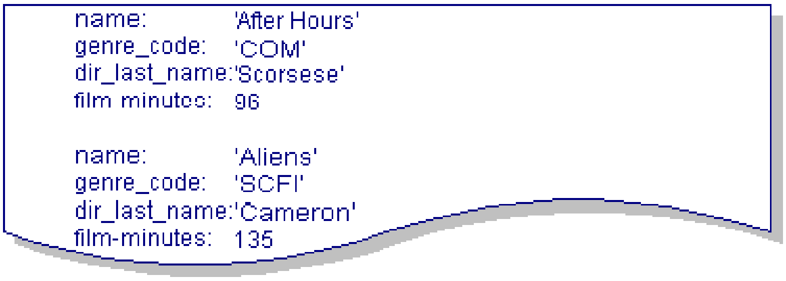
```
title_id:        56
name:            'After Hours'
genre_code:      'COM'
dir_last_name:   'Scorsese'
dir_first_name:  'Martin'
film-minutes:    96
rating_code:     'R'
release_date:    1986/01/01  00:00:00
pricecat:        'G'
```

However, this syntax is not recommended for use inside an application. It is recommended that you include a select list in a `SELECT` statement.

Specifying a Select List

A select list determines which columns will be included in the result set. In the following example, the select list contains the name, `genre_code`, `dir_last_name` and `film_minutes` columns:

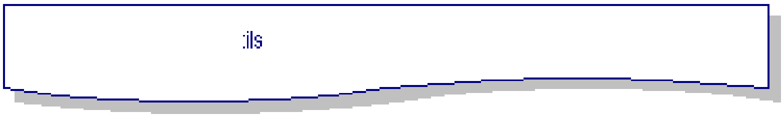
```
SELECT name, genre_code, dir_last_name, film_minutes
   FROM titles;
```



```
name:          'After Hours'
genre_code:    'COM'
dir_last_name:'Scorsese'
film minutes:  96

name:          'Aliens'
genre_code:    'SCFI'
dir_last_name:'Cameron'
film-minutes:  135
```

The select list can also include aggregate functions.

```
SELECT AVG (film_minutes) FROM titles;
```

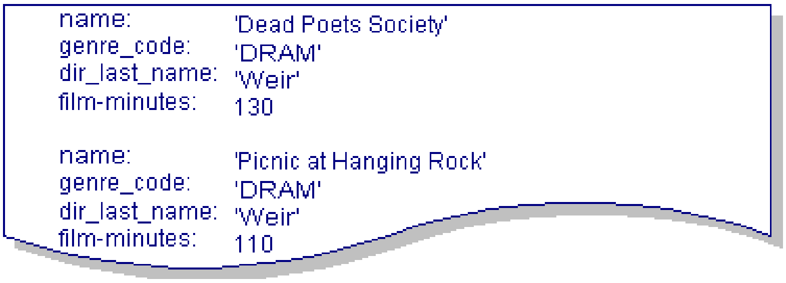


```
:ils
```

Specifying a `WHERE` Clause

You can choose which rows will be a part of the result set by including a `WHERE` clause:

```
SELECT name, genre_code, dir_last_name, film_minutes
   FROM titles WHERE dir_last_name = 'Weir';
```



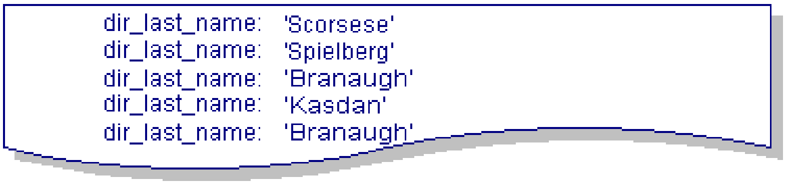
```
name:           'Dead Poets Society'
genre_code:     'DRAM'
dir_last_name:  'Weir'
film-minutes:   130

name:           'Picnic at Hanging Rock'
genre_code:     'DRAM'
dir_last_name:  'Weir'
film-minutes:   110
```

There are other search conditions available. Refer to page 6-54 for information on the `WHERE` clause.

Obtaining Unique Entries

You can include only unique rows in the result set by specifying the keyword `DISTINCT`. The following statement gets a list of directors:

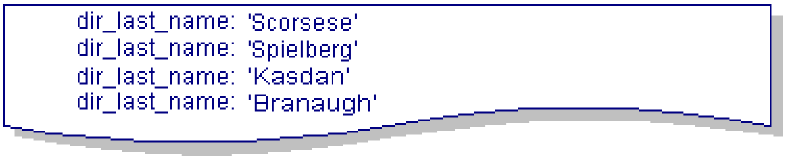
```
SELECT dir_last_name FROM titles;
```



```
dir_last_name:  'Scorsese'
dir_last_name:  'Spielberg'
dir_last_name:  'Branaugh'
dir_last_name:  'Kasdan'
dir_last_name:  'Branaugh'
```

By using `DISTINCT`, the duplicate names are excluded from the result set:

```
SELECT DISTINCT dir_last_name FROM titles;
```

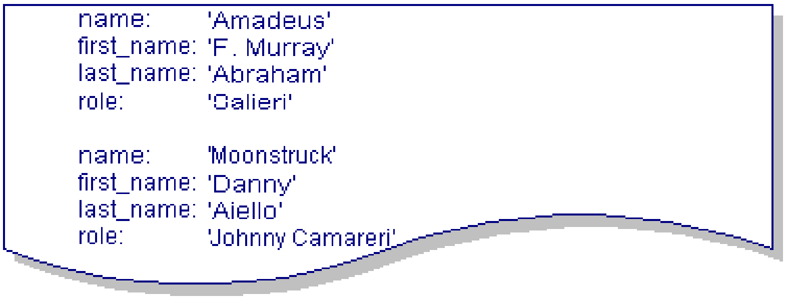


```
dir_last_name: 'Scorsese'
dir_last_name: 'Spielberg'
dir_last_name: 'Kasdan'
dir_last_name: 'Branaugh'
```

Obtaining Data from Multiple Tables

You can obtain information from more than one database table by using joins:

```
SELECT name, first_name, last_name, role
    FROM actors, titles, roles
    WHERE titles.title_id = roles.title_id
    AND actors.actor_id = roles.actor_id;
```



```
name:       'Amadeus'
first_name: 'F. Murray'
last_name:  'Abraham'
role:       'Salieri'

name:       'Moonstruck'
first_name: 'Danny'
last_name:  'Aiello'
role:       'Johnny Camareri'
```

Refer to page 6-27 for more information on joins.

See Also BETWEEN Predicate, Joins, GROUP BY Clause, HAVING Clause, Subqueries, WHERE Clause

Subqueries

Nest a SELECT statement within another statement

Description

In a subquery, you can nest a `SELECT` statement within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. The main syntax restriction is that a subquery cannot contain an `ORDER BY` clause. Refer to the sections on each keyword for any additional syntax restrictions. The subquery is enclosed in parentheses. Many statements using subqueries can alternatively be constructed using joins.

There are five keywords used for subqueries: `EXISTS`, `IN`, `ANY`, `ALL`, and `SOME`. These are explained below.

`EXISTS`

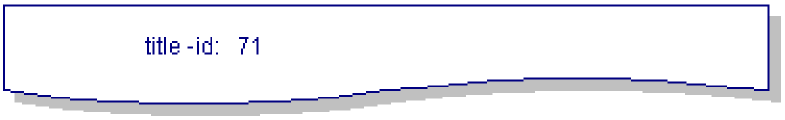
```
WHERE [NOT] EXISTS (subquery)
```

The `EXISTS` keyword tests for the presence of a result set from the subquery. The subquery can contain one or more columns. Since you are testing to see if any rows are returned, you can use `SELECT *` instead of a select list in the subquery.

If the `NOT` keyword is also specified, the `WHERE` clause is satisfied if there are no rows in the result set.

The following query checks to see if all of the video titles have entries in the roles table:

```
SELECT title_id FROM titles WHERE NOT EXISTS
       (SELECT * FROM roles
       WHERE roles.title_id = titles.title_id);
```



```
title -id:  71
```

`IN`

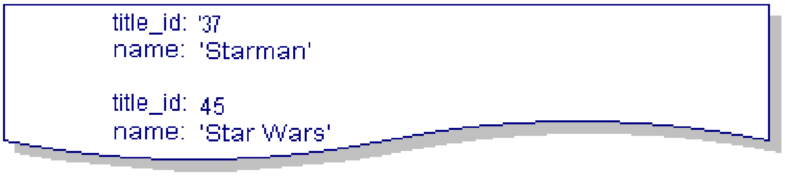
```
WHERE expression [NOT] IN (subquery)
```

The `IN` subquery condition evaluates whether the expression in the `WHERE` clause matches any row returned in the subquery. The subquery using `IN` can only return one column, but it can return more than one row.

A subquery using the keyword IN is equivalent to the same subquery using =ANY.

The following query lists which science fiction movies a customer has previously rented:

```
SELECT title_id, name FROM titles WHERE genre_code = 'SCFI'
       AND title_id IN
       (SELECT title_id FROM rentals
       WHERE cust_id = 6);
```



```
title_id: '37
name:  'Starman'

title_id: 45
name:  'Star Wars'
```

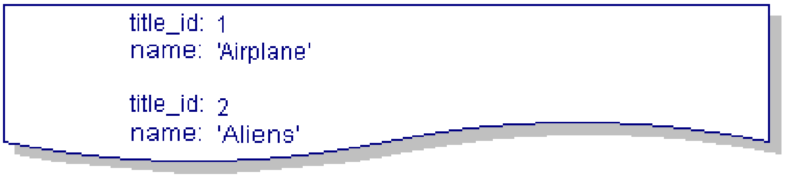
ANY, ALL, SOME

WHERE expression comparison-operator ANY (subquery)WHERE expression comparison-operator ALL (subquery)WHERE expression comparison-operator SOME (subquery)

The keywords ANY, ALL, or SOME are used with a subquery with one of the following comparison operators: >, >=, <, <=, <>, or =.

A subquery using ANY or SOME tests to see if the comparison is true for at least one of the values returned by the subquery. If the subquery returns no value, the search condition is false.

```
SELECT title_id, name FROM titles WHERE title_id = ANY
       (SELECT title_id FROM tapes WHERE status = 'I');
```

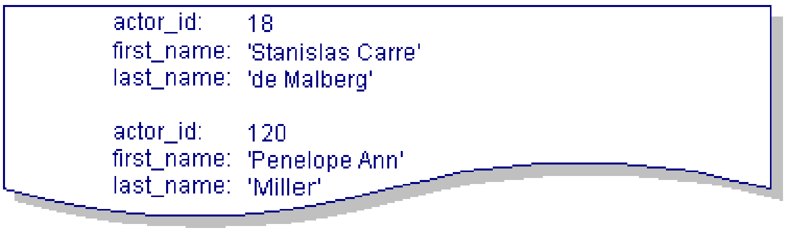


```
title_id: 1
name:  'Airplane'

title_id: 2
name:  'Aliens'
```

A subquery using ALL tests to see if the comparison is true for every value returned by the subquery. If the subquery returns no value, the search condition is true as well.

The following query tests to see which actors are not in the roles table:

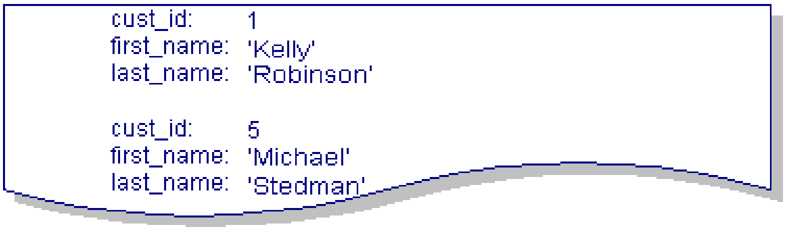
```
SELECT actor_id, first_name, last_name FROM actors
       WHERE actor_id <> ALL
       (SELECT actor_id FROM roles);
```



```
actor_id:    18
first_name:  'Stanislas Carre'
last_name:   'de Malberg'

actor_id:    120
first_name:  'Penelope Ann'
last_name:   'Miller'
```

The keywords ANY, ALL, or SOME can be omitted if you know that the subquery will return exactly one value. The following example returns one value by using an aggregate function. This query finds the customers whose rental amount was higher than average:

```
SELECT cust_id, first_name, last_name FROM customers
       WHERE rent_amount >
       (SELECT AVG(rent_amount) FROM customers);
```



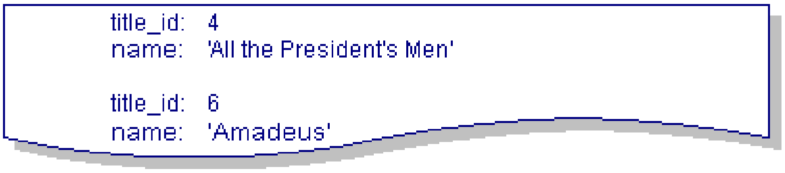
```
cust_id:     1
first_name:  'Kelly'
last_name:   'Robinson'

cust_id:     5
first_name:  'Michael'
last_name:   'Stedman'
```

Nested Subqueries

A subquery can also contain another subquery.

The following query finds the videos depicting dramatic stories that are also in the rentals table:

```
SELECT title_id, name FROM titles WHERE title_id IN
    (SELECT title_id FROM rentals WHERE title_id IN
    (SELECT title_id FROM titles WHERE genre_code = 'DRAM'));
```



```
title_id:   4
name:       'All the President's Men'

title_id:   6
name:       'Amadeus'
```

See Also   Joins

UPDATE Statement

Update information in a database table

```
UPDATE table-name SET column-name = value [, ...]
   [WHERE search-conditions]
```

table-name
: Unique identifier for the database table.

SET
: The SET clause lists both the columns to be updated and the new values for those columns.

column-name
: Name of the column to be modified.

WHERE
: The WHERE clause specifies which rows will be updated. Refer to page 6-54 for more information on the WHERE clause.

search-conditions
: Specifies the conditions for the selection of data. For more information, refer to page 6-54.

Description

The UPDATE statement modifies the value of one or more columns in the specified table.

If the UPDATE statement is part of a transaction, the update can be undone by rolling back the transaction.

Warning: If you omit the WHERE clause, all rows in the table are updated.

Examples

The following statement increases each price category by 10%. Since there is no WHERE clause, this statement updates each row in the pricecats table:

```
UPDATE pricecats SET price = price * 1.1;
```

The following statement updates the price category for a video:

```
UPDATE titles SET pricecat = 'G' WHERE title_id = 57;
```

The following query updates the member status to the frequent renter category if a customer rents over 10 videos a month:

```
UPDATE customers SET member_status = 'F'
   WHERE cust_id IN
   (SELECT cust_id FROM rentals
   WHERE rental_date
   BETWEEN '1993/09/01 00:00:00' AND '1993/10/01 00:00:00'
   GROUP BY cust_id HAVING COUNT(*) > 10 ');
```

See Also WHERE Clause

WHERE Clause

Specify search conditions and/or specify the relationship between tables

```
WHERE search-conditions
WHERE column-name join-operator column-name
```

---

search-conditions
: Specifies the conditions for the selection of data.

column-name
: Specifies a column in each of the tables to be joined.

join-operator
: Specifies the join operator. Refer to page 6-27 for more information about joins.

Description

The `WHERE` clause performs two functions:

- Specifying the search conditions for the result set.
- Specifying the connection between tables named in the `FROM` clause.

A result set contains only the rows in the database that meet the search conditions. If more than one search condition is included in a `WHERE` clause, connect the conditions with the logical operators `AND` or `OR`.

Search Conditions

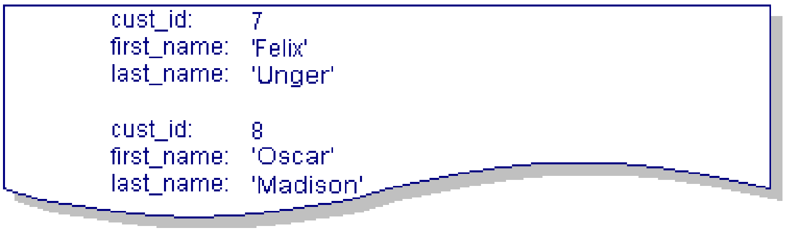
Search conditions can include the following:

`BETWEEN`

```
WHERE [NOT] expression [NOT] BETWEEN expression AND
expression
```

The `BETWEEN` predicate specifies a range of database values. The following statement returns all the customers who joined during a certain year:

```
SELECT cust_id, first_name, last_name FROM customers
       WHERE member_date BETWEEN '1992/01/01 00:00:00'
       AND '1993/01/01 00:00:00';
```



```
cust_id:      7
first_name:   'Felix'
last_name:    'Unger'

cust_id:      8
first_name:   'Oscar'
last_name:    'Madison'
```

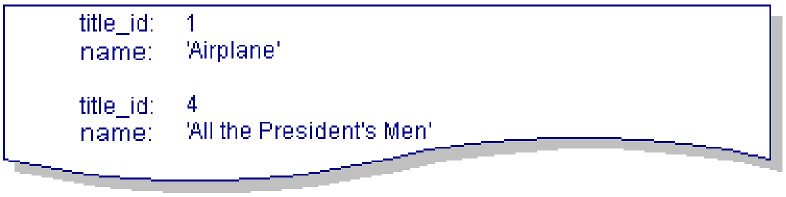
`EXISTS`

`WHERE [NOT] EXISTS subquery`

The `EXISTS` keyword tests for the presence of a result set from the subquery. If the `NOT` keyword is also specified, the `WHERE` clause is satisfied if there are no rows in the result set. The subquery is enclosed in parentheses.

Notice that the subquery uses an * instead of a select list since you are merely testing whether rows meet the subconditions specified in the query.

```
SELECT title_id, name FROM titles WHERE EXISTS
       (SELECT * FROM tapes WHERE title_id = tapes.title_id
       AND status = 'I');
```



```
title_id:   1
name:       'Airplane'

title_id:   4
name:       'All the President's Men'
```

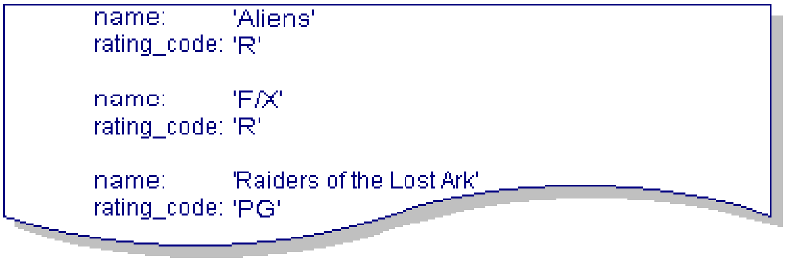
`IN`

`WHERE expression [NOT] IN subquery`
`WHERE expression [NOT] IN values-list`

The `IN` keyword evaluates whether or not the expression in the `WHERE` clause matches a row in the subquery or a value in the values list. The subquery using `IN` can only return one column, but it can return more than one row.

The following query uses a values list to find the adventure and science fiction videos. It tests whether the `genre_code` for each video matches `ADV` or `SCFI`.

```
SELECT name, rating_code FROM titles
       WHERE genre_code IN ('ADV', 'SCFI');
```



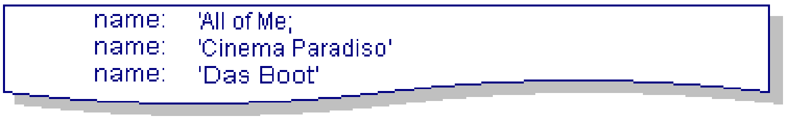
```
name:        'Aliens'
rating_code: 'R'

name:        'F/X'
rating_code: 'R'

name:        'Raiders of the Lost Ark'
rating_code: 'PG'
```

IS NULL

WHERE column-name IS [NOT] NULL

The keyword `IS NULL` searches for null values in the specified column.

```
SELECT name FROM titles WHERE rating_code IS NULL;
```



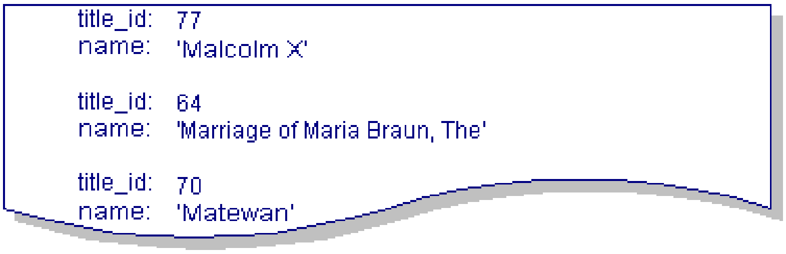
```
name:   'All of Me;
name:   'Cinema Paradiso'
name:   'Das Boot'
```

LIKE

WHERE column-name [NOT] LIKE literal [ESCAPE literal]

A `LIKE` predicate selects rows where a column value matches a specified pattern. The following query finds the video titles that begin with "M."

```
SELECT title_id, name FROM titles WHERE name LIKE 'M%'
```



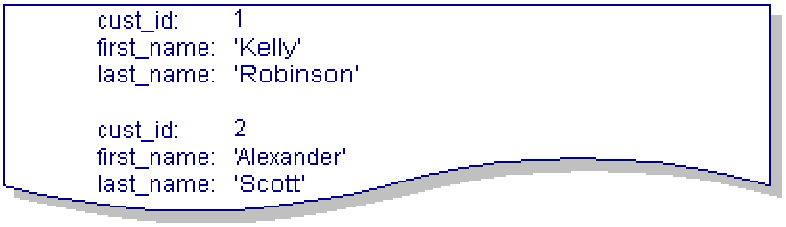
```
title_id:   77
name:       'Malcolm X'

title_id:   64
name:       'Marriage of Maria Braun, The'

title_id:   70
name:       'Matewan'
```

Operators

WHERE expression {> | < | >= | <= | = | <>}{expression | subquery}

Operators allow you to compare column values. The following query finds the customers who have rented more than 2000 videos. For more information on using operators in subqueries, refer to page 6-48.

```
SELECT cust_id, first_name, last_name FROM customers
       WHERE num_rentals > 200;
```

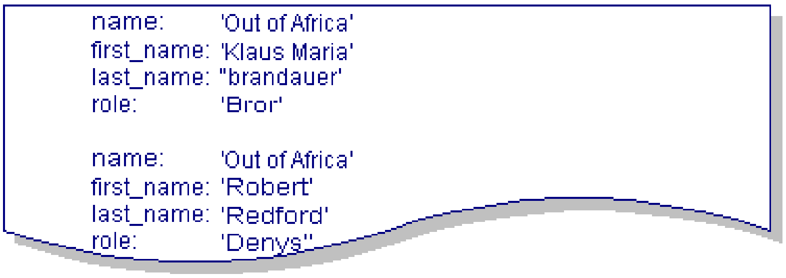


```
cust_id:     1
first_name:  'Kelly'
last_name:   'Robinson'

cust_id:     2
first_name:  'Alexander'
last_name:   'Scott'
```

Specifying Joins

The `WHERE` clause also specifies the interconnecting columns between tables in joins. The following statement illustrates a multiple join. For additional information on joins, refer to page 6-27.

```
SELECT name, first_name, last_name, role
    FROM titles, actors, roles
    WHERE titles.title_id = roles.title_id
    AND roles.actor_id = actors.actor_id
    AND titles.title_id = 62;
```



```
name:       'Out of Africa'
first_name: 'Klaus Maria'
last_name:  "brandauer'
role:       'Bror'

name:       'Out of Africa'
first_name: 'Robert'
last_name:  'Redford'
role:       'Denys"
```

See Also   BETWEEN Predicate, Joins, LIKE Predicate, Null Values, Operators, Subqueries

SQL Syntax Summary

```
CREATE DATABASE database-name

CREATE TABLE table-name (
   column-name data-type [(length)] [NOT NULL] [, column-name ...]
   [PRIMARY KEY (column-name, column-name ...]), ]
   [UNIQUE (column-name [, column-name ...]), ]
   [FOREIGN KEY (column-name [, column-name ...])
   REFERENCES table-name (column-name [, column-name ...]) [,]
      ]

CREATE TABLE table-name (
   column1 data-type [(length)] NOT NULL,
   column2 data-type [(length)] NOT NULL,
   column3 data-type [(length)] NOT NULL,
   column4 data-type [(length)] NOT NULL,
   column5 data-type [(length)],
   column6 data-type [{length)],
   PRIMARY KEY (column1, column2),
   UNIQUE (column2),
   UNIQUE (column3, column4),
   FOREIGN KEY (column5, column6)
      REFERENCES table2 (column, column),
   FOREIGN KEY (column5) REFERENCES table3 (column)
   )

DELETE FROM table-name [WHERE search-conditions]

DROP DATABASE database-name

DROP TABLE table-name

INSERT INTO table-name [(column-list)]
   VALUES (literal| NULL [, ...])

INSERT INTO table-name [column-name [, ...]] query-expression

SELECT [DISTINCT] {select-list | *}
   FROM table-name [correlation-name] [, ...]
   [WHERE search-conditions]
   [GROUP BY [correlation-name.]column-name [, ...]]
   [HAVING search-conditions]
   [ORDER BY {integer| [correlation-name.]column-name} [, ...] ]

UPDATE table-name SET column-name = value [, ...]
   [WHERE search-conditions]
```

APPENDIX

A JDB Utilities

This chapter describes the utilities available with JDB:

- `isql`—A command-line interactive SQL utility
- `jdbroll`—Updates a database using its journal files
- `jisql`—Graphical interactive SQL editor
- `mksql`—Outputs SQL statements for the specified database
- `tbldata`—Imports/exports data to and from a database

isql

Access a command line interactive SQL utility

```
isql databaseName
```

databaseName
Specifies the name of the database or `@system` if a database isn't available.

Description

The `isql` utility is a command-line interactive editor for JDB that lets you to execute any database statement. It is provided as a command-driven alternative to the JISQL graphical environment described in Chapter 5, "Using JISQL."

While `isql` and JISQL are similar in many respects, they are not identical. JISQL has a series of macro commands that are not available in `isql`, so JISQL scripts containing these commands will not run under `isql`. Although the SQL commands in `isql` end with a `;` as a termination character, all JISQL commands terminate with a `;`. Use the interactive SQL capability described here only if you want to bypass the JISQL environment.

Starting ISQL

To start `isql`, at the command line, type:

```
$SMBASE/jdb/bin/isql databaseName
```

where *`databaseName`* is the name of an existing database or `@system` if a database isn't available. The screen displays the following numeric prompt where you can enter any JDB statement: `1>`

Creating a New Database

If this is your first JDB session, or if you want to create a new database, first start `isql`. Generally, on UNIX systems, it is located in `$SMBASE/jdb/bin`.

```
isql
```

```
At the prompt, logon to
```

```
JDB using
@system:
logon @system
```

Once you are connected, create a new database by typing:

`create database` *`databaseName`*`;`

`databaseName` must conform to the file naming conventions of the operating system.

To connect to the new database in order to create database tables and enter data, enter:

`logon` *`databaseName`*

Executing SQL Statements

You can execute any SQL statement available in JDB by ending each database statement with a `;` as the command terminator. For example:

```
1> SELECT title_name, name, genre_code FROM titles;
```

A line without a trailing semi-colon is concatenated with the next line until a semicolon is reached. Therefore, one statement can span multiple lines.

Executing ISQL Statements

Table A-1 lists the commands available in `isql`. These commands allow you to edit a query, read in a query file, or execute an operating system command.

Note: These commands do not end with a semi-colon. Also, in order for these commands to be recognized, each must start on the first character of a command line.

Table A-1 Commands for isql

Command	Syntax	Description
`clear`	`clear`	Empties the input buffer. Command must start in the first column of a new line.
`comment`	#	Specifies that the line is a comment.

Table A-1 Commands for isql *(Continued)*

Command	Syntax	Description
`commit`	`commit`	Saves additions and edits you make to the database since the last commit or rollback or since connecting to the database.
`edit`	`edit`	Starts an editing shell for entering statements. The editing program is determined by the environment variable `EDITOR` or `SMEDITOR`.
`exit`	`exit`	Exits `isql`.
`list`	`list`	Displays last executed command.

`logon`	`logon` *`databaseName`*
	`logon` *`@system`*
		Connects to another database file or `@system`.

output	output *`filename`*	Redirects output to a file. If you specify output without a file name, it redirects output to the screen.
`quit`	`quit`	Quits `isql`.
`read`	`read` *`filename`*	Reads and executes the SQL commands in a text file. To execute more than one command, each command must end with a semi-colon (`;`).
`rollback`	`rollback`	Undoes all additions and edits made to the database since the last commit or rollback or since connecting to the database.
`system`	`system` *`commandName`*	Executes the named operating system command.

Exiting ISQL

To exit `isql`, type:

```
exit
```

An automatic commit is generated when you exit the `isql` session using either the `quit` or `exit` commands. Specify `rollback` if you do not want to keep your database changes.

jdbroll

Restore a transaction log

```
jdbroll databaseName journalName [journalName ...]
```

`databaseName`
Specifies the name of the database.

`journalName`
Specifies the name of the journal file(s).

Description

The `jdbroll` utility allows you to update the database based on your log files.

When you logon to a database for the first time, JDB creates a journal file named *`j1databaseName`*. For example, a database having the `videobiz` would have a journal file named `j1videobiz`. The next time you log on, the information in the current file (`j1videobiz`) is copied to the file *`j0databaseName`*. If the file already exists, it is overwritten.

mksql

Translate an existing JDB database into its CREATE TABLE and INSERT statements

```
mksql databaseName
```

`databaseName`
Specifies the name of the database.

Description The `mksql` utility uses an existing JDB database to output a set of SQL statements from which the database could be rebuilt. For each table, it writes a `CREATE TABLE` statement, followed by a series of `INSERT` statements for the data in the table.

Example The following result set illustrates a portion of the output for the actors table in the videobiz database.

```
create table actors (
                    actor_id long not null
                    last_name character(25) not null
                    first_name character(20)
                    primary key (actor_id)
);

insert into actors (
                    actor_id
                    last_name
                    first_name)
values (
                    1
                    'Abraham'
                    'F. Murray'
);
```

tbldata

Read rows in a database table to/from text files

```
tbldata [-d delimiter] -x exportFile databaseName tableName
tbldata [-d delimiter] -i importFile databaseName tableName
```

-d *delimiter*
: Specifies the column delimiter. The delimiter character might need to be enclosed in quotation marks. For example, to specify a space as the delimiter:

```
tbldata -d " " -x exportFile databaseName tableName
```

If no delimiter is specified, `tbldata` uses `TAB` as the delimiter.

-x
: Logs onto the specified database and writes each row of the specified table to the specified text file.

-i
: Logs onto the specified database and inserts each row of the specified text file into the specified table.

exportFile
: Name of the text file where the data will be written.

importFile
: Name of the text file containing the data to be inserted into JDB.

databaseName
: Name of the JDB database.

tableName
: Name of the database table.

Description

The `tbldata` utility can be used two different ways:

- With the `-x` argument to convert rows in a database table to a text file.
- With the `-i` argument to insert rows into a database table from a text file.

With these options, you must specify both the database and the database table.

When using `-i` option, the database table must already exist. Also, the column values must be listed in the same order as the columns in the database table.

APPENDIX

B JDB-Specific Error Messages

This chapter lists the error messages that can occur while using JDB. The messages are stored in the Panther message file.

If an error occurs using the `isql` utility, the error message is displayed on the screen. If the error prompt appears followed by numbers instead of error message text, check the setting of the variable `SMVARS`.

If an error occurs in an application, the message that appears on the screen depends on the type of error handler currently installed. There are DBMS commands and global variables available in Panther's database drivers for use in an error handler. For more information, refer to Chapter 37, "Processing Application Errors," in *Application Development Guide.*

Error Message Listing

```
Aggregate function not allowed in current context
(DM_JDB_AGGREGATE_NOT_ALLOWED)
```

Cause:

Aggregate function appears in the wrong context.*

Action:

Aggregate functions can appear in the select list of a `SELECT` statement or in a `HAVING` clause.

`Ambiguous column reference (DM_JDB_AMBIGUOUS_COLUMN_REF)`

Cause:

In a multiple join, a column name has been specified without its corresponding table name.

Action:

Add the table name to the column references.

`Bad Input (DM_JDB_BAD_INPUT)`

Cause:

Data formatted incorrectly for `tbldata` utility.*

Action:

Edit input file. For information on data types, refer to page 6-15.

`Corrupt JDB Database detected (DM_JDB_DB_CORRUPT)`

Action:

Exit the database, restart that same database, and reissue the statement to see if the message disappears. If not, use `tbldata` to unload the database. Check the ASCII file before reloading the database.

`Current cursor is not attached to a database (DM_JDB_NODB)`

Cause:

Executing a query or data modification statement while connected to the system catalog or while not connected to a database.

Action:

Logon to the desired database, and re-execute the command.

`Duplicate column assignment (DM_JDB_DUP_COL_ASSIGNMENT)`

Cause:

The column has been specified twice in the `SET` clause of an `UPDATE` statement.

Action:

Edit statement and eliminate duplicate setting.

`Duplicate column name (DM_JDB_DUP_CNAME)`

Cause:

The column name has already been specified for that table.

Action:

Assign each column in the database table a unique name.

`Duplicate table alias (DM_JDB_DUPTABLEALIAS)`

Cause:

Using the same table alias for more than one table.

Action:

Assign each table alias a unique name.

`Duplicate table name (DM_JDB_DUP_TNAME)`

Cause:

Creating a database table that matches an existing table name.

Action:

Assign each table in the database a unique name.

`File I/O Error (DM_JDB_FILE_IO_ERR)`

Cause:

1.)Database is not in the current directory. 2) Database does not exist. 3) Database name was misspelled.

Action:

Depending on the desired outcome, either specify the database path or create the database in the current directory.

`Internal datatype conversion failed (DM_JDB_CONVERSION_FAILED)`

Cause:

Inserting a character string into a datetime column in an `INSERT` or `UPDATE` statement. Inserting a `;` instead of a `:` when specifying a datetime value.

Action:

Check to see if the values match the data types of the columns.

`Invalid Database/Table Handle (DM_JDB_BAD_HANDLE)`

Cause:

Internal error in opening and closing table structure.*

Action:

Exit database and restart.

`Invalid Table operation (DM_JDB_INVALID_TABLE_OP)`

Cause:

Trying to update system tables.

Action:

Only `SELECT` statements can be used on the system tables.

`Journal error (DM_JDB_JOURNAL_ERROR)`

Cause:

Database is read-only.

Action:

Change the file permissions.

`Key columns must be specified as not null (DM_JDB_KEY_MUST_BE_NULL)`

Cause:

Primary key column was specified in the `CREATE TABLE` statement without the `NOT NULL` keywords.

Action:

Insert the `NOT NULL` keywords for primary key columns.

`Maximum record length exceeded (DM_JDB_MAX_RECLEN_EXCEEDED)`

Cause:

Row definition is greater than 1K.

Action:

Edit table definition to a maximum of 1024 bytes for each row.

`More than one primary key was specified (DM_JDB_MULT_PKEY)`

Cause:

An additional `PRIMARY KEY` clause was specified in the `CREATE TABLE` statement.

Action:

Specify one `PRIMARY KEY` clause using commas to separate the primary key columns.

`Must close Database first (DM_JDB_DATABASE_OPEN)`

Cause:

Attempting to drop a database while that database connection is still active.

Action:

Logon to another database or to the system catalog in order to drop the database.

`Must drop Database first (DM_JDB_DATABASE_EXISTS)`

Cause:

Attempting to create a database when that database already exists.

Action:

Depending on the desired outcome, either 1) drop the database so that it can be recreated, or 2) use another database name.

`Not implemented (DM_JDB_NOT_IMPLEMENTED)`

Cause:

Feature not implemented in JDB.*

Action:

n/a

`Read-Only handle (DM_JDB_READONLY)`

Cause:

Database file is specified to be read-only.

Action:

This appears as a warning when you log on and as an error if you attempt to insert or update data in the database.

`NULL not allowed (DM_JDB_NULL_NOT_ALLOWED)`

Cause:

Column has been defined in the `CREATE TABLE` statement as `NOT NULL`.

Action:

In an `INSERT` statement, a value must be entered for all columns defined as `NOT NULL`.

`Syntax error (DM_JDB_SYNTAX_ERROR)`

Cause:

Character strings are not enclosed in single quotation marks in an `INSERT` statement.

Action:

Add quotation marks, and reissue statement.

Cause:

Reserved keyword used as a table or column name in a `CREATE TABLE` statement.

Action:

Change table or column name.

`Table not found (DM_JDB_TABLE_NOT_FOUND)`

Cause:

Database table does not exist.

Action:

Create table, or query `systabs` for table names in the database.

`Temporary database error (DM_JDB_TMPDATABASE_ERR)`

Cause:

Unable to create temporary database needed for processing.

Action:

Check memory available.*

`The number of values specified does not equal the number of columns (DM_JDB_INVALID_VALUES_COUNT)`

Cause:

In an `INSERT` statement, the number of columns in the column list and the number of column values in the values list is not the same.

Action:

Check `INSERT` statements.

`The subquery returned too many rows (DM_JDB_SUBQ_TOO_MANY_ROWS)`

Cause:

Subquery returned multiple rows when statement needs one value.

Action:

Edit query to use different search conditions.

`Type mismatch (DM_JDB_TYPE_MISMATCH)`

Cause:

Inserting a character string into a column specified as integer.

Action:

JDB performs the insertion converting the character string to `0`. Edit the statement to the correct value.

`Unresolved column reference (DM_JDB_UNRESOLVED_COLUMN_REF)`

Cause:1

Misspelled column names in SQL statements. 2) Column values not enclosed in single quotes. 3) Column listed in an `ORDER BY` clause is not in the select list. 4) Incorrect table name included in correlation name or alias.

Action:

Correct syntax and reissue statement.

APPENDIX

C Keywords in JDB

This chapter lists the keywords specified in the ANSI standard for SQL. These keywords cannot be used in JDB expressions and are, therefore, not available for use as table, column or database names.

all	alter	and
any	as	asc
authorization	avg	
begin	between	by
char	character	check
clear	close	cobol
commit	continue	count
create	current	cursor
database	datetime	dec
decimal	declare	default
delete	desc	distinct
double	drop	
edit	end	escape
exec	exists	exit

fetch	float	for
foreign	fortran	found
from		
go	goto	grant
group		
having		
in	indicator	insert
int	integer	into
is	is null	
key		
language	like	list
logon	long	
max	min	module
not	null	numeric
of	on	open
option	or	order
output		
pascal	pli	precision

primary	precision	primary
privileges	procedure	public
quit		
read	real	references
rollback		
save	schema	section
select	set	smallint
some	sql	sqlcode
sqlerror	sum	system
table	to	
union	update	user
using		
values	view	
whenever	where	with
work		

APPENDIX

D Videobiz Database

This section describes the database tables in the videobiz database. The following information is listed for each table:

- Column names
- Data type of each column.
- Length of character columns
- Status of column detailing whether it is a primary or foreign key and whether it can accept null values
- Description of the data to be entered into the column
- Sample entry

Videobiz Schema

The following tables outline the database tables in the videobiz database.

Table D-1 Actors table

Column Name	Data Type	Length	Status	Description	Sample
`actor_id`	integer		primary key not null	Unique number code for each actor.	87

Table D-1 Actors table *(Continued)*

Column Name	Data Type	Length	Status	Description	Sample
`last_name`	char	25	not null	Actor's last name or only name.	Ullmann
`first_name`	char	20		Actor's first name.	Liv

Table D-2 Codes table

Column Name	Data Type	Length	Status	Description	Sample
`code_type`	char	32	primary key not null	Type of code. Corresponds to column name.	`genre_code`
`code`	char	4	primary key not null	Code value.	`ADV`
`dscr`	char	40		Description of code value.	Adventure

Table D-3 Customers table

Column Name	Data Type	Length	Status	Description	Sample
`cust_id`	integer		primary key not null	Unique number code for each customer.	2
`last_name`	char	25	not null	Customer's last name.	Scott
`first_name`	char	20	not null	Customer's first name.	Alexander
`address1`	char	40		Customer's address.	5601 Wilson
`address2`	char	40		Additional address information.	
`city`	char	25		City customer lives in.	Geneva
`state_prov`	char	10		State/Province.	NY

Table D-3 Customers table *(Continued)*

Column Name	Data Type	Length	Status	Description	Sample
`postal_code`	char	10		Postal code.	10234
`phone`	char	15		Customer's telephone number.	515-221-4111
`cc_code`	char	4		Code for type of credit card. List in codes table.	VISA
`cc_number`	char	16		Number on credit card.	4000...
`cc_exp_month`	integer			Month of credit card expiration. 1=January, 12=December.	2
`cc_exp_year`	integer			Year of credit card expiration (4 digits).	1994
`member_date`	datetime			Date when customer became a member.	1991/05/30 00:00:00
`member_status`	char	1	not null	Current status of membership. Values include: (A)ctive, (I)nactive, (F)requent renter.	A
`num_rentals`	integer		not null	Total number of rentals customer has made.	105
`rent_amount`	float		not null	Total amount of money paid by customer.	175.00
`notes`	char	254		Comments about customer.	Likes ADV videos.

Table D-4 Flag table

Column Name	Data Type	Length	Status	Description	Sample
`yesno`	char	1		Flag used in the sample application.	Y

Table D-5 Pricecats table

Column Name	Data Type	Length	Status	Description	Sample
`pricecat`	char	1	primary key not null	Unique letter code for each category.	N
`pricecat_dscr`	char	40		Category description.	New Release
`rental_days`	integer		not null	Number of rentals days available in this category.	2
`price`	float		not null	Amount to be paid for rentals in this category.	2.50
`late_fee`	float		not null	Amount of late fee for rentals in this category.	2.00

Table D-6 Rentals table

Column Name	Data Type	Length	Status	Description	Sample
`cust_id`	integer		primary key foreign key not null	Code identifying the customer for this rental.	3
`title_id`	integer		primary key foreign key* not null	Code identifying the video title for this rental.	69

Table D-6 Rentals table *(Continued)*

Column Name	Data Type	Length	Status	Description	Sample
`copy_num`	integer		primary key foreign key not null	Copy of this video being rented.	2
`rental_date`	datetime		primary key not null	Date/time the video was rented.	1993/10/29 19:56:00
`due_back`	datetime		not null	Date the video is due back to avoid late fee.	1993/11/01 00:00:00
`return_date`	datetime			Actual date/time the video was re turned; NULL until then.	NULL
`price`	float		not null	Rental fee for video at time rental was made.	3.50
`late_fee`	float		not null	Late fee per day for video at time rental was made.	1.00
`amount_paid`	float		not null	Total amount paid on this rental as of current date.	3.50
`rental_status`	char	1	not null	Status of rental. Values include (C)urrently out, Back and (P)aid, (B)alance is due.	C
`rental_comment`	char	76		Comments about rental, if any.	NULL
`modified_date`	datetime		not null	Date this record was last modified.	1993/10/29 19:56:00
`modified_by`	integer		foreign key not null	Last user who modified record.	2

**`title_id`* *is a foreign key from the tapes table, in combination with* *`copy_num`*.

Table D-7 Roles table

Column Name	Data Type	Length	Status	Description	Sample
`title_id`	integer		primary key foreign key not null	Unique number code for each video title.	33
`actor_id`	integer		primary key foreign key not null	Unique number code for each actor.	87
`role`	char	40		Role the actor plays in the video.	Marianne

Table D-8 Tapes table

Column Name	Data Type	Length	Status	Description	Sample
`title_id`	integer		primary key foreign key not null	Unique number code for each video title.	33
`copy_num`	integer		primary key not null	Number identifying the copy of this video.	1
`status`	char	1	not null	Code specifying the current status of this copy. Values include (A)vailable, (R)eserved, (O)ut, (I)nactive.	O
`times_rented`	integer		not null	Number of times this copy has been rented.	53

Table D-9 Titles table

Column Name	Data Type	Length	Status	Description	Sample
`title_id`	integer		primary key not null	Unique number code for each video title.	33

Table D-9 Titles table *(Continued)*

Column Name	Data Type	Length	Status	Description	Sample
`name`	char	60	not null	Video title.	Scenes from a Marriage
`genre_code`	char	4		Code specifying the video category. Values include: `ADLT, ADV, CHLD, CLAS, COM, HORR, MUS, MYST, SCFI, TV, VID`. See codes table.	`CLAS`
`dir_last_name`	char	25		Director's last name.	Bergman
`dir_first_name`	char	20		Director's first name.	Ingmar
`film_minutes`	integer			Length of the video.	168
`rating_code`	char	4		Rating code given the film by the Motion Picture Association of America. Values include: `G, PG, PG13, R, NC17`. See codes table.	PG
`release_date`	datetime			Year the film was released to movie theatres.	1974/01/01 00:00:00
`pricecat`	char	1	foreign key not null	Code taken from the pricecats table specifying the price category.	G

Table D-10 Title_dscr table

Column Name	Data Type	Length	Status	Description	Sample
`title_id`	integer		primary key foreign key not null	Unique number code for each video title.	33

Table D-10 Title_dscr table *(Continued)*

Column Name	Data Type	Length	Status	Description	Sample
`line_no`	integer		primary key not null	Line number of the video description.	1
`dscr_text`	char	76		Description of the video.	Relationship of a couple...

Table D-11 Users table

Column Name	Data Type	Length	Status	Description	Sample
`user_id`	integer		primary key not null	Unique number code for each system user/employee.	3
`logon_name`	char	8		User's logon name.	jack
`password`	char	8		User's password.	go
`last_name`	char	25		User's last name.	Ryan
`first_name`	char	20		User's first name.	Jack
`customer_flag`	char	1		Y allows access to customer subsystem.	Y
`admin_flag`	char	1		Y allows access to administrative subsystem.	N
`marketing_flag`	char	1		Y allows access to marketing subsystem.	Y
`frontdesk_flag`	char	1		Y allows access to front desk subsystem.	Y

Index