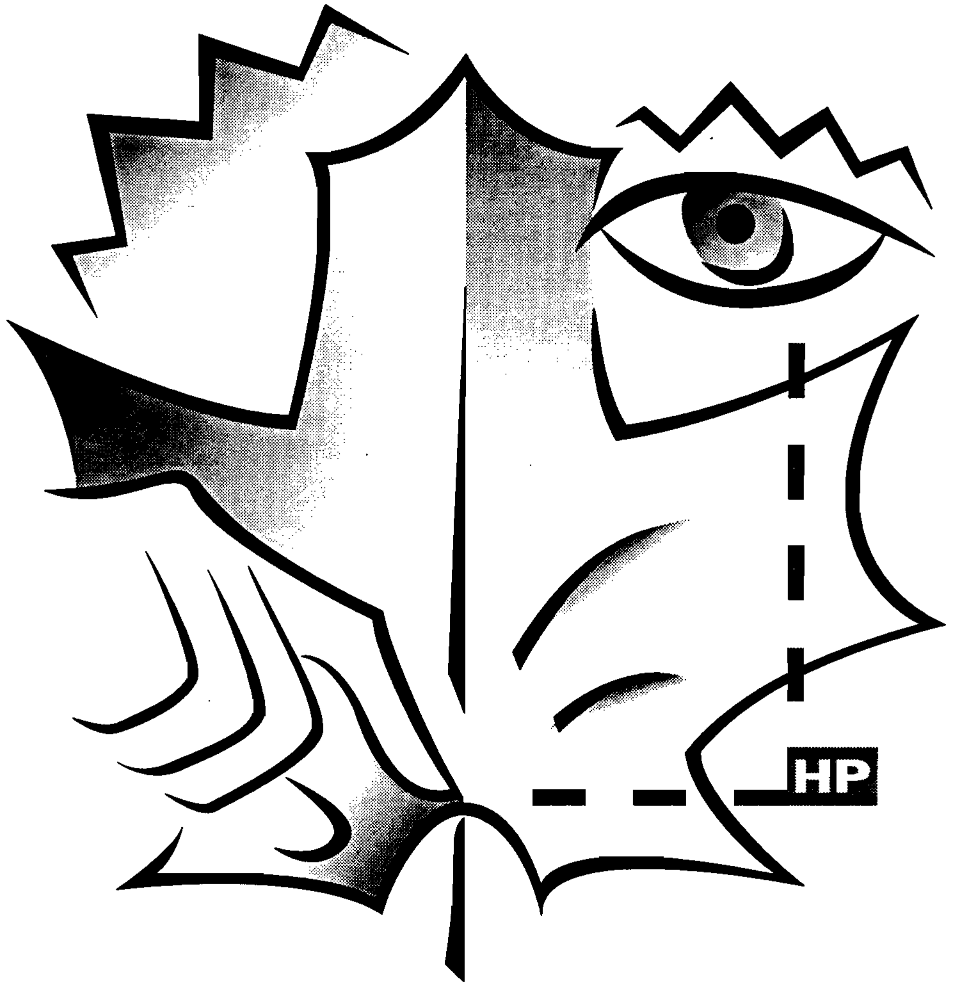
Interex '95

Conference & Expo



14–18 August 1995
Toronto, Ontario, Canada

Workbook

T300

Tutorial: IMAGE/SQL: Issues and Answers Concerning SQL Tables

Leslie-Anne Bain
Hewlett-Packard Co.

interex
Shared Knowledge. Shared Power.

The International Association of Hewlett-Packard Computing Professionals

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

IMAGE/SQL:

Issues and Answers Concerning SQL Tables

Leslie-Anne Bain
Software Design Engineer
Hewlett Packard, CSY Division

Presentation #T300

© COPYRIGHT HEWLETT-PACKARD CO. 1995

1

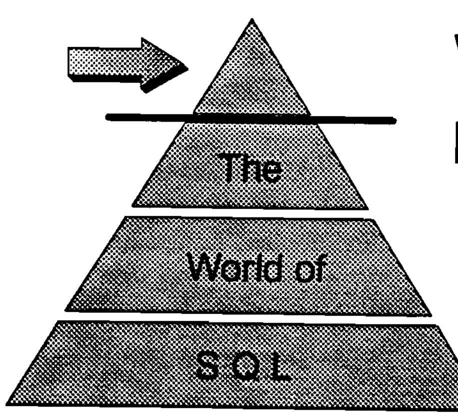
Overview

Purpose of Slide

To explain the subjects to be covered during this talk.

What am I going to learn?

- Key concepts needed to develop an SQL application that accesses both TurboIMAGE datasets and SQL tables.



WARNING! This class will provide an SQL "jump-start" (be aware that much information will follow)!

Key Points

If you are a TurboIMAGE programmer, you are already familiar with the key concepts needed to develop an application that calls TurboIMAGE intrinsics. The goal of this tutorial is to make you familiar with similar concepts for the SQL part of IMAGE/SQL, so that you have the ability to develop an SQL application that accesses information stored in either TurboIMAGE data sets or SQL tables.

I'll go out on a limb here, but I'm willing to bet you did not learn everything that you needed to know about TurboIMAGE from a two-hour tutorial. When I was involved with IMAGE programming, I took several multi-day customer courses to come up to speed.

SQL is complex. Where does one start? This tutorial will try to give you a jump-start into the world of SQL. My objective is to expose you to key SQL concepts, and to show you where to go for additional information. The illustration on the diagram is supposed to be similar to an iceberg. We'll cover the 'tip' of the iceberg in this class, but not the entire iceberg.

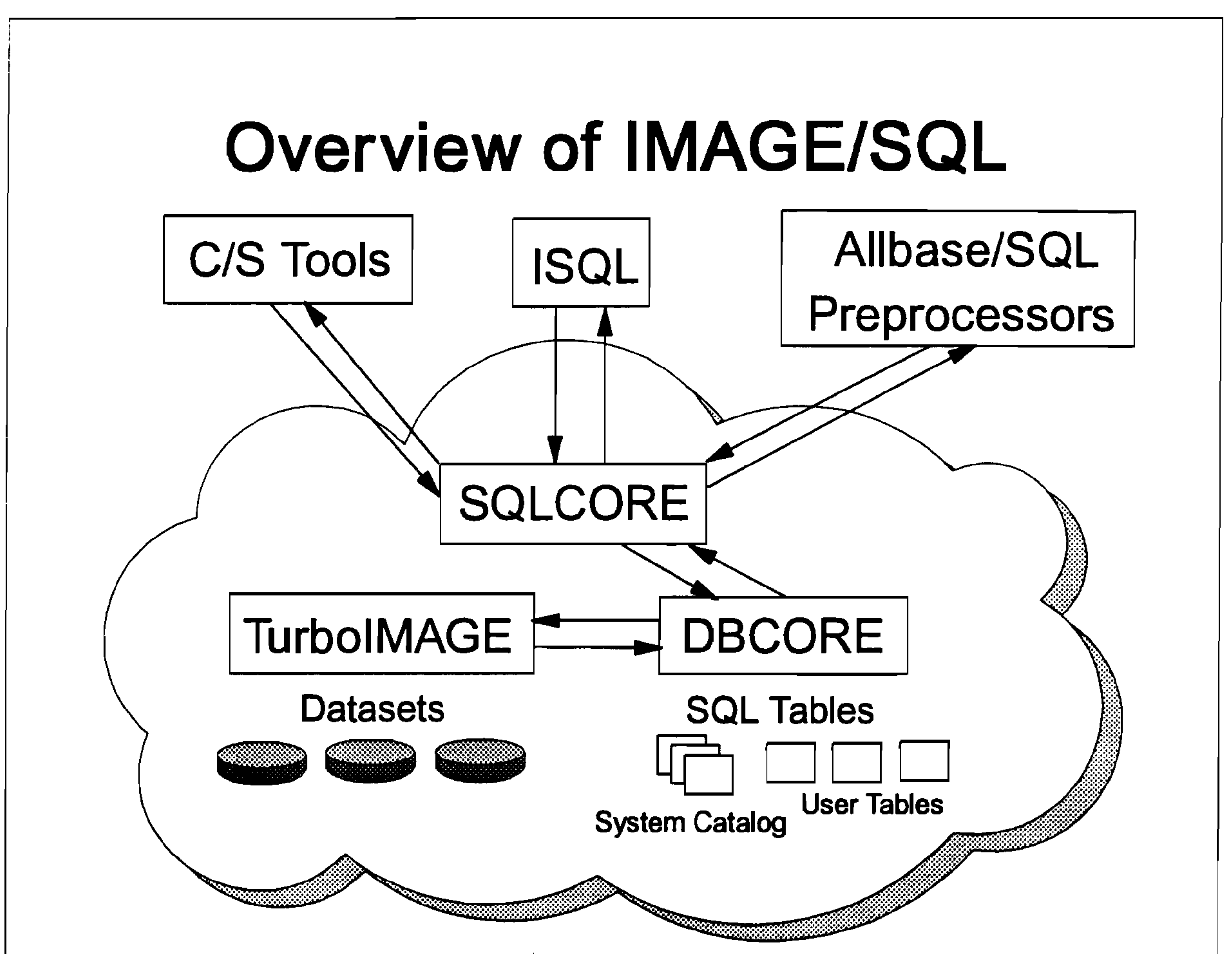
When you jump-start your car, an intense burst of energy passes into the battery. In this tutorial you will be exposed to much information. I recommend that you try to focus on the "big picture" instead of all of the details. Save this workbook for later reference.

Major topics to be covered:

- **Chapter 1** - Overview
  - How are TurboIMAGE/XL, ALLBASE/SQL, and IMAGE/SQL related?
  - Why should I care about SQL tables?
  - What SQL terminology should I know?
  - Where can I get additional information?
- **Chapter 2** - SQL Storage and Security
  - Review physical storage for TurboIMAGE
  - Explain how SQL tables are physically stored.
  - Explain how SQL security is implemented.
- **Chapter 3** - Trouble-Shooting with SQLMON
  - What is it?
  - Where is it?
  - How do I use it?
- **Chapter 4** - SQL Transactions and Locking
  - SQL Transactions (starting, ending, timeouts, etc.)
  - How are deadlocks resolved by ALLBASE/SQL?
  - SQL DBECon file parameters associated with locking.
  - How does the SQL optimizer affect locking?
  - What is an SQL cursor?
  - And everything else you ever wanted to know about SQL locking!

Purpose of Slide

To explain how TurboIMAGE/XL, ALLBASE/SQL, and IMAGE/SQL are related.



G.1 includes Turbo Image indexing info.

Key Points

IMAGE/SQL consists of three components:

1. **TurboIMAGE/XL** is a set of programs and procedures for defining, creating, maintaining and accessing TurboIMAGE/XL databases.
   - A database is a collection of *data sets.*
   - Each data set contains *data entries*, also known as records.
   - The physical storage manager is known as TurboIMAGE.
   - When writing a program to access data from the database, the developer includes calls to TurboIMAGE intrinsics.
   - The developer controls the data access path. For example, the developer controls whether a *chained read* or a *serial read* is used.
2. **ALLBASE/SQL** is a set of programs and procedures for creating, maintaining, and accessing ALLBASE/SQL DBEnvironments.
   - A DBEnvironment is a collection of SQL tables.
   - Each SQL table contains *rows*, also known as *tuples.*
   - The physical storage manager is known as DBCORE.
   - When writing a program to access data from the DBEnvironment, the developer includes industry-standard SQL statements.
   - The developer does not control the data access path. A second manager known as SQLCORE creates an *access plan* for each SQL query, and performs the appropriate sequence of calls to DBCORE.
   - SQLCORE uses information in a special set of SQL tables known as the *system catalog* when it generates the access plan. If the system catalog shows that an index exists on a table, then SQLCORE will probably ask DBCORE to perform an *index scan* instead of a *serial scan.*
3. Additional software (including the IMAGESQL utility) that registers information about a TurboIMAGE/XL database into the system catalog of an ALLBASE/SQL DBEnvironment. To SQLCORE, the TurboIMAGE data sets look like SQL tables, with one exception. SQL statements that update, insert, or delete data ultimately get routed to the TurboIMAGE storage manager when TurboIMAGE data sets are referenced.

   As a result, the large suite of client/server tools that support ALLBASE/SQL now support TurboIMAGE/XL too!

The SQL part of IMAGE/SQL is ALLBASE/SQL. From an SQL point of view, the major difference between IMAGE/SQL and ALLBASE/SQL is the amount of data you can store in user SQL tables. In IMAGE/SQL the limit is 12 MegaBytes (3000 4K pages). In ALLBASE/SQL there is no limit.

Purpose of Slide

Why should I care about SQL tables?

Why should I care about SQL tables?

- I may not need to!
- I may want to evolve to SQL because it is an industry standard interface.
- I may benefit from using the SQL tools or approach for program development.
- I may use SQL tables without realizing it!
- I may decide to create an SQL table instead of a new data set.
- IMAGE/SQL gives me more options. The final choice is up to me!

Key Points

- You may not need to care. Don't waste your time reading this document if you have already decided you will only use IMAGE/SQL and some off-the-shelf SQL decision support tool (such as *Information Access* by Hewlett-Packard) to access your existing TurboIMAGE data sets in "read-only" mode.

  If you don't insert data into the SQL side (either the system catalog or user tables), then you don't need to worry about SQL storage management or concurrency issues (except at IMAGESQL ATTACH and DETACH time).

  You might want to read Chapter 2 to learn more about SQL security, however, as future enhancements to IMAGE/SQL security may take advantage of the SQL approach.

- You may be considering evolving your existing TurboIMAGE applications to SQL because it is "open":

  ```
  "[IMAGE/SQL] is a superb migration tool, which will allow users to develop
  new applications on a true fully-featured relational database (ALLBASE/SQL),
  while still accessing the data held in IMAGE legacy applications.  This
  means a migration that once looked unavoidable and daunting suddenly can be
  carried out little by little, over several years, to suit your company's
  convenience ... HP has pulled off its biggest coup since Compatibility Mode
  gave us painless migration to PA-RISC."
        Martin Knapp, PROACTIVE SYSTEMS SENIOR SOFTWARE ENGINEER
        Reprinted from the December, 1994 edition of InterexPress.
  ```

  ALLBASE/SQL is available on both the HP3000 and the HP9000. In addition, ALLBASE/SQL adheres to industry SQL standards, which means you can eventually "migrate" to another SQL vendor (and hardware platform) if you ever want to.

- You may be considering evolving to SQL because of the benefits of using a relational model instead of a network model:

  - Client/Server development tools.
  - The DBA has more flexibility to make database changes. Programs don't "break" when changes are made because they don't have intimate knowledge about how data is physically stored.

- If you decide to develop an SQL application that will access the data in your TurboIMAGE database, you will probably insert compiled SQL statements (known as stored sections) into the system catalog (a special set of SQL tables). The system catalog serves essentially the same function as the root file (it contains metadata about each of the objects in the DBEnvironment). SQLCORE accesses information in the system catalog when it executes most SQL statements. Locks are acquired during this process, which can result in concurrency problems if you are not aware of the issues.

- And finally, you may want to create an SQL table instead of creating a new data set in a TurboIMAGE schema.

Purpose of Slide

What SQL terminology should I know?

What are the buzzwords?

TurboIMAGE/XL	IMAGE/SQL,ALLBASE/SQL
■ Database (1)	■ DBEnvironment
■ Database (2)	■ Set of tables having the same owner.
■ Data Set	■ Table
■ Data Entry,Record	■ Row,Tuple
■ Field,Item	■ Column
■ Chain,Key	■ Index
■ Passwords	■ Grant/Revoke to a User/Group.
■ DBXBEGIN,DBXEND	■ Transaction

Key Points

- The concept of a *database* in TurboIMAGE maps to two different concepts in IMAGE/SQL and ALLBASE/SQL:
  1. A *DBEnvironment* (or DBE) is primarily a set of SQL tables that share the same system catalog and the same logging and recovery. Typically you store and restore all files in the DBE at the same time.
  2. In SQL terminology, a *database* is a set of SQL tables within a particular DBEnvironment that share the same owner. When a table is created, both an *owner* and a *tablename* are assigned to completely identify it. Since it is possible to create tables having different owners within a single DBE, multiple databases can exist within a DBE.

     IMAGE/SQL allows you to attach several TurboIMAGE databases to a single DBEnvironment. The data sets are registered as tables in the system catalog; all data sets from the same TurboIMAGE database have the same owner. So, each TurboIMAGE database is also a unique "database" in SQL terminology.
- An SQL table is similar to a TurboIMAGE data set.
- An SQL row has one or more columns.
- You can create hash indexes, B-tree indexes or parent-child relationships (referential constraints) on SQL tables.
- In SQL, security is primarily based on the end-user's logon id, not on a password. The DBA grants and revokes database priviledges to a particular logon id. The DBA can also define a *group*, and grant/revoke to the group in the same way as to a particular logon id. Each logon id added to the group has the same priviledges as the group. Finally, the DBA can add groups to other groups.
- In TurboIMAGE, several types of transactions are defined. A *dynamic* transaction is a logical transaction which has the following attributes:
  1. It begins with a DBXBEGIN call and normally ends with a DBXEND call; it can be rolled back dynamically with a DBXUNDO call.
  2. It spans only one database. A program can open more than one database if it needs to access data sets from different databases, and can have dynamic transactions in effect on different databases at the same time.

  In DBCORE, only one type of transaction is defined. A DBCORE transaction is essentially equivalant to a TurboIMAGE dynamic transaction, and has the following attributes:
  1. It begins with a BEGIN WORK statement and normally ends with a COMMIT WORK statement; it can be rolled back dynamically with a ROLLBACK WORK statement.
  2. It spans only one database. A program that needs to access tables in more than one DBEnvironment can do so by connecting to multiple DBEnvironments; however, only one connection can have an active transaction at any time (it is not possible to reference tables from multiple DBEnvironments within a single transaction). A program must end a transaction on one DBEnvironment before starting a transaction on a second DBEnvironment.

     Most interactive Client/Server tools only allow you to connect to a single DBEnvironment at a time.

Purpose of Slide

Where can I get additional information?

IMAGE/SQL and TurboIMAGE Documentation Product Numbers

Table 1-1. IMAGE/SQL and TurboIMAGE Documentation Product Numbers

CATEGORY	PRODUCT NUMBER	TITLE
IMAGE/SQL and TurboIMAGE	30391-90001 36385-90008 36385-90001	TurboIMAGE/XL Database Management System Reference Manual Getting Started with HP IMAGE/SQL HP IMAGE/SQL Administration Guide
PC API	B2463-90013	Read Me Before Installing HP ALLBASE/SQL PC API

ALLBASE/SQL Documentation Product Numbers

Table 1-2. ALLBASE/SQL MPE/iX Documentation Product Numbers

CATEGORY	PRODUCT NUMBER	TITLE
General Reference	36389-90011 36216-90001 36216-90009 36216-90096 92534-90011	Up and Running with ALLBASE/SQL ALLBASE/SQL Reference Manual ALLBASE/SQL Message Manual ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL HP ALLBASE/QUERY User's Guide
Database and Network Administration	36216-90031 36216-90005 36216-90102	ALLBASE/NET User's Guide ALLBASE/SQL Database Administration Guide ALLBASE/SQL Performance and Monitoring Guidelines
Embedded SQL Programming Guides	36216-90100 36216-90023 36216-90006 36216-90030 36216-90007	ALLBASE/SQL Advanced Application Programming Guide ALLBASE/SQL C Application Programming Guide ALLBASE/SQL COBOL Application Programming Guide ALLBASE/SQL FORTRAN Application Programming Guide ALLBASE/SQL Pascal Application Programming Guide
REPLICATE	B2494-90002	HP ALLBASE Replicate User's Guide
PC API	36216-90104	HP PC API User's Guide for ALLBASE/SQL and IMAGE/SQL

Table 1-3. ALLBASE/SQL HP-UX Documentation Product Numbers

CATEGORY	PRODUCT NUMBER	TITLE
General Reference	36389-90011 36217-90001 36217-90009 36217-90188 92534-64001	Up and Running with ALLBASE/SQL ALLBASE/SQL Reference Manual ALLBASE/SQL Message Manual ALLBASE/ISQL Reference Manual HP ALLBASE/QUERY User's Guide
Database and Network Administration	36217-90093 36217-90005 36217-90185	ALLBASE/NET User's Guide ALLBASE/SQL Database Administration Guide ALLBASE/SQL Performance and Monitoring Guidelines
Embedded SQL Programming Guides	36217-90186 36217-90014 36217-90058 36217-90013 36217-90007	ALLBASE/SQL Advanced Application Programming Guide ALLBASE/SQL C Application Programming Guide ALLBASE/SQL COBOL Application Programming Guide ALLBASE/SQL FORTRAN Application Programming Guide ALLBASE/SQL Pascal Application Programming Guide
REPLICATE	B3480-90002	HP ALLBASE Replicate User's Guide
PC API	36217-90187	HP PC API User's Guide for ALLBASE/SQL

2

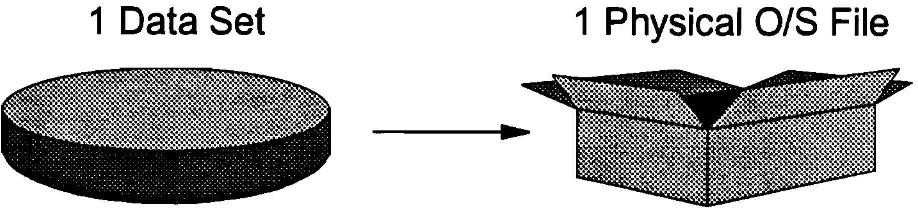
SQL Storage and Security

Purpose of Slide

To review physical storage for TurboIMAGE/XL.

Physical Storage of TurboIMAGE Data Sets

Traditional Model



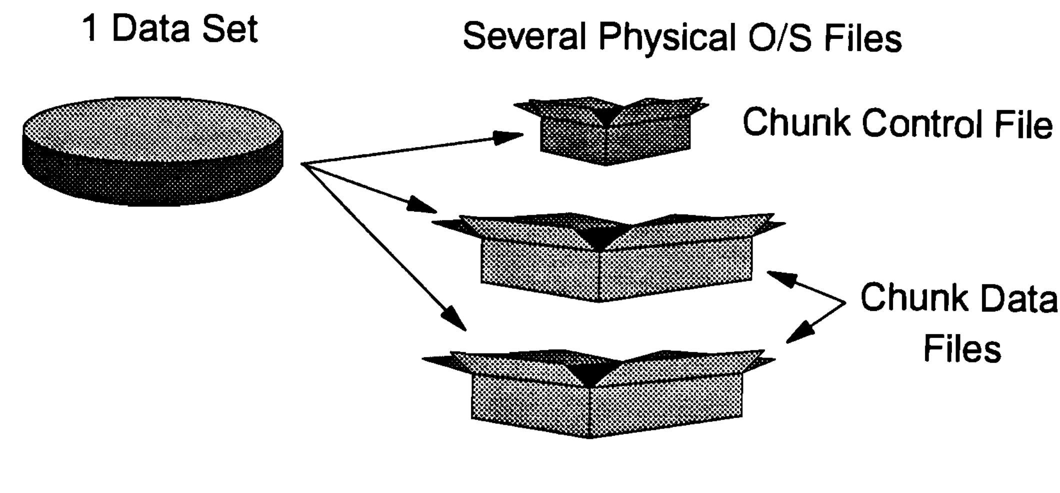
Max: 4 GigaBytes

Key Points

- Traditional limit is 4 GigaBytes per data set.
- Limit exists because there is a one-to-one mapping between data sets and MPE files. Each data set is stored in its own MPE file.

Purpose of Slide

To review Jumbo Data Sets in TurboIMAGE/XL.

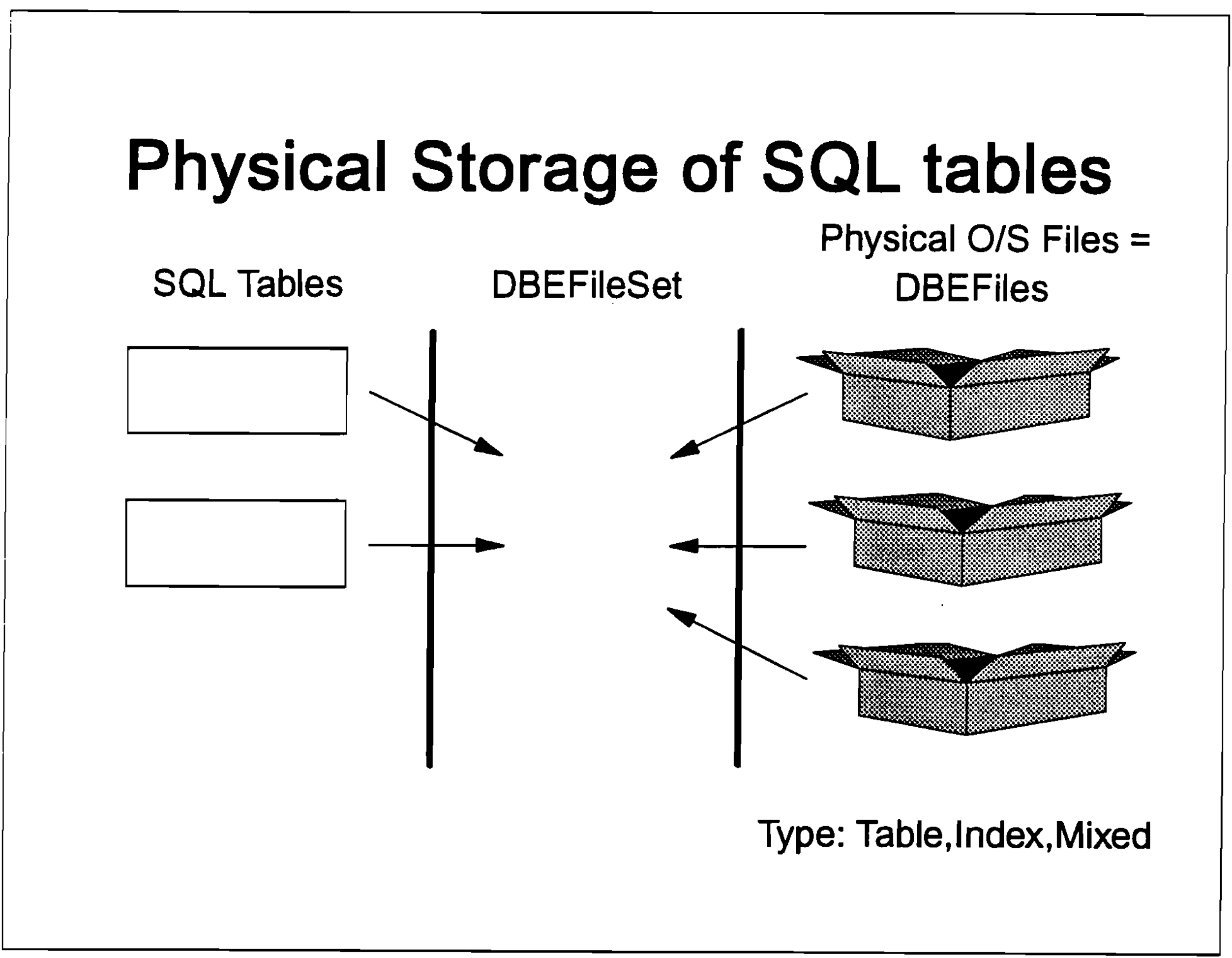


Key Points

- With jumbo data sets, there is a one-to-many mapping between data sets and MPE files. Each data set can be stored in several MPE files.
- There is one *chunk control file* for each jumbo data set (filecode -408). This file has no user data, and is typically small. It has information about the *chunk data files*, and is used for mapping record numbers to particular files.
- There can be up to 10 chunk data files for each jumbo data set (filecode -409). Each file is typically 4 GigaBytes, except the last one, which may be smaller. The data format within the chunk data files is identical to the format used in traditional data sets.

Purpose of Slide

To explain how SQL tables are physically stored.



Key Points

- ALLBASE/SQL allows a one-to-one, a one-to-many, or a many-to-many mapping between SQL tables and MPE files. Typically, customers use a many-to-many mapping.
- Physical files are known as *DBEFiles*. You use the CREATE DBEFILE and ADD DBEFILE statements to add physical files to a DBEFileSet:

```
isql => CREATE DBEFILESET partsfileset;

isql => CREATE DBEFILE parts2 WITH PAGES=253, NAME='parts2',
> TYPE=mixed;

isql => ADD DBEFILE parts2 TO DBEFILESET partsfileset;
isql => commit work;
```

  There is virtually no limit to the number of DBEFiles that can be added to a DBEFileSet.
- When you use the CREATE TABLE statement, you specify the DBEFileSet that you want the table to belong to:

```
isql => CREATE PUBLIC TABLE Parts.Vendors
> (PartNumber char(16),
>  VendorNumber INTEGER)
> IN partsfileset;
isql => commit work;
```

  Rows for the table can be stored on any of the DBEFiles that have been added to the DBEFileset.
- B-tree indexes are always created in the same DBEFileset as the table.
- DBEFiles come in three flavors: TABLE, INDEX, and MIXED. Data rows for SQL tables can only be inserted onto TABLE or MIXED DBEFiles. Index data can only be inserted onto INDEX or MIXED DBEFiles. An error is returned if you try to insert data and there is no space available on any of the appropriate DBEFiles.
- To improve performance, you can put INDEX and DATA DBEFiles on different disc drives.
- DBEFiles can be created so that they dynamically expand. When you use the CREATE DBEFILE statement, you can specify an initial size, a maximum size, and an increment value (that is, the number of pages to add each time the file needs to expand).

Purpose of Slide

To explain how DBEFileSets are used in ALLBASE/SQL.

What is a DBEFileSet?

- A set of DBEFiles.
- A layer of indirection that allows SQL tables to become virtually unlimited in size.
- Rows for an SQL table can be stored on any DBEFile(s) that belong to the same DBEFileSet as the table.
- If a table grows large, space can easily be added by simply adding another DBEFile to the DBEFileSet.

Key Points

- When the DBEnvironment is created, a special DBEFileset known as the SYSTEM DBEFileset is created.
  - Initially, the SYSTEM DBEFileset has a single DBEFile known as DBEFile0 in it.
  - The system catalog tables reside in the SYSTEM DBEFileset.
  - You can add additional DBEFiles to the SYSTEM DBEFileset.
  - You can also add additional tables to the SYSTEM DBEFileset, but it is better to put user tables into other DBEFileSets instead.
- To remove a DBEFile from a DBEFileset, use the REMOVE DBEFILE and DROP DBEFILE statements:

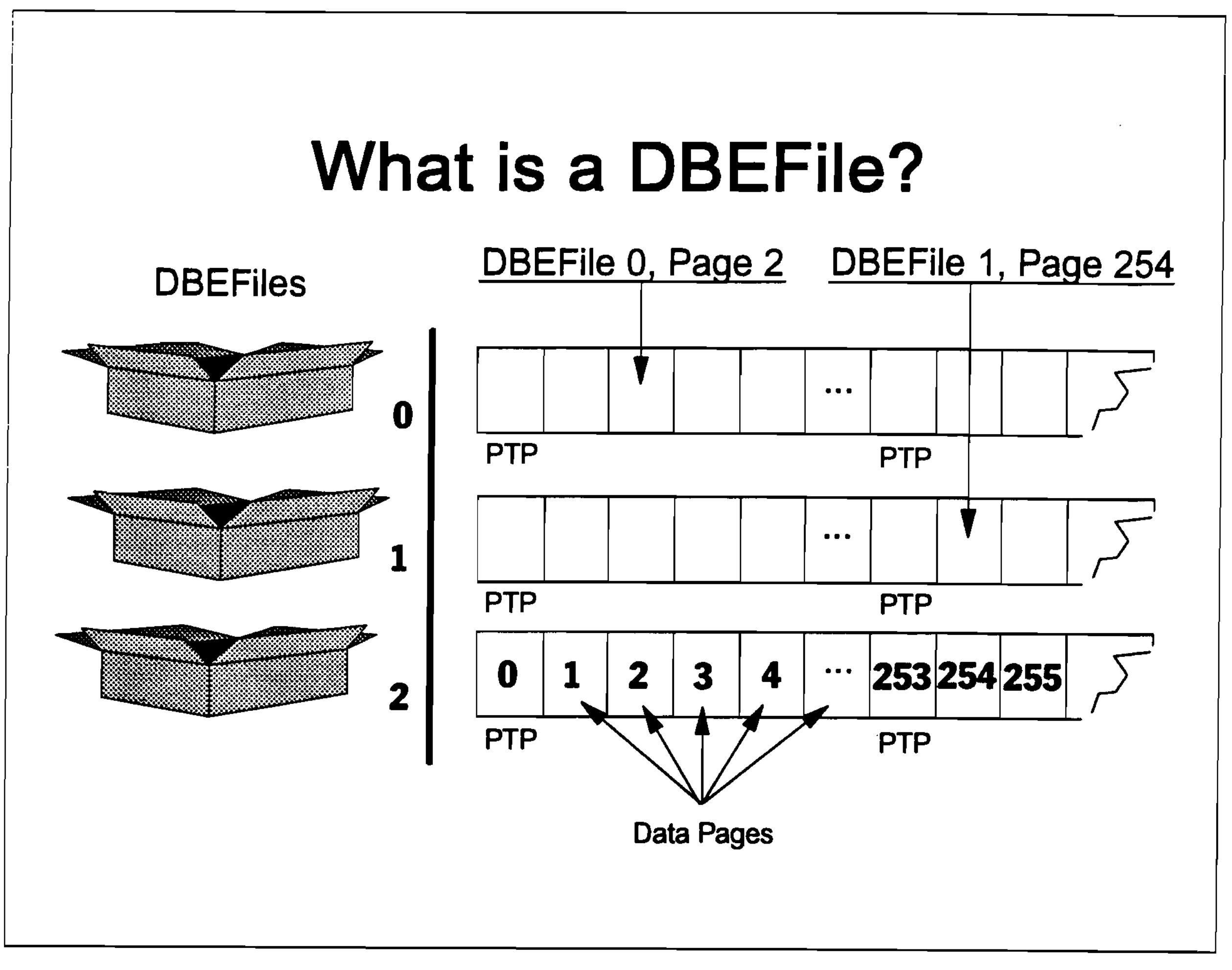
```
isql => REMOVE DBEFILE parts2 FROM DBEFILESET partsfileset;

isql => DROP DBEFILE;
isql => commit work;
```

  The DBEFile must be empty.
- To delete the definition of a DBEFileSet, use the DROP DBEFILSET statement;

```
isql => DROP DBEFILESET partsfileset;
isql => commit work;
```

Purpose of Slide

To explain the internals of a DBEFile.



Key Points

- DBEFiles are also called *chunks*.
- Each DBEFile has a number, which is visible under the DBEFNUMBER column of SYSTEM.DBEFILE.

```
isql => select DBEFNUMBER,* from SYSTEM.DBEFILE order by DBEFNUMBER;
```

- In TurboIMAGE, each data set file can be thought of as a set of records. Each record in the file corresponds to a record in the data set.
- In ALLBASE/SQL, each DBEFile can be thought of as a set of 4K pages. Each page has a number. There are basically two types of pages, **page table pages** and **data pages**. Each type of page has its own layout.
  - A page table page (PTP) is basically a table of contents for the next 252 pages. For every 253 pages in the DBEFile, the first page is a page table page, and the other 252 pages are data pages.
  - A data page is used to store table data or index data, depending on the type of the DBEFile. Rows for an SQL table are always stored on data pages, in a special format understood by DBCORE. Up to 256 rows can be stored on a single page, depending on the size of the rows.
- Each row inserted onto a page has a physical address (also known as the TID, or tuple id). The form of a TID is

```
(DBEFile Number):(Page Number):SLOT.
```

  Each row on the page has a different SLOT number (from 0 to 255).

  The TID for a row on the page highlighted in DBEFile 0 might be: 0:2:128.

  The TID for a row on the page highlighted in DBEFile 1 might be: 1:254:7.

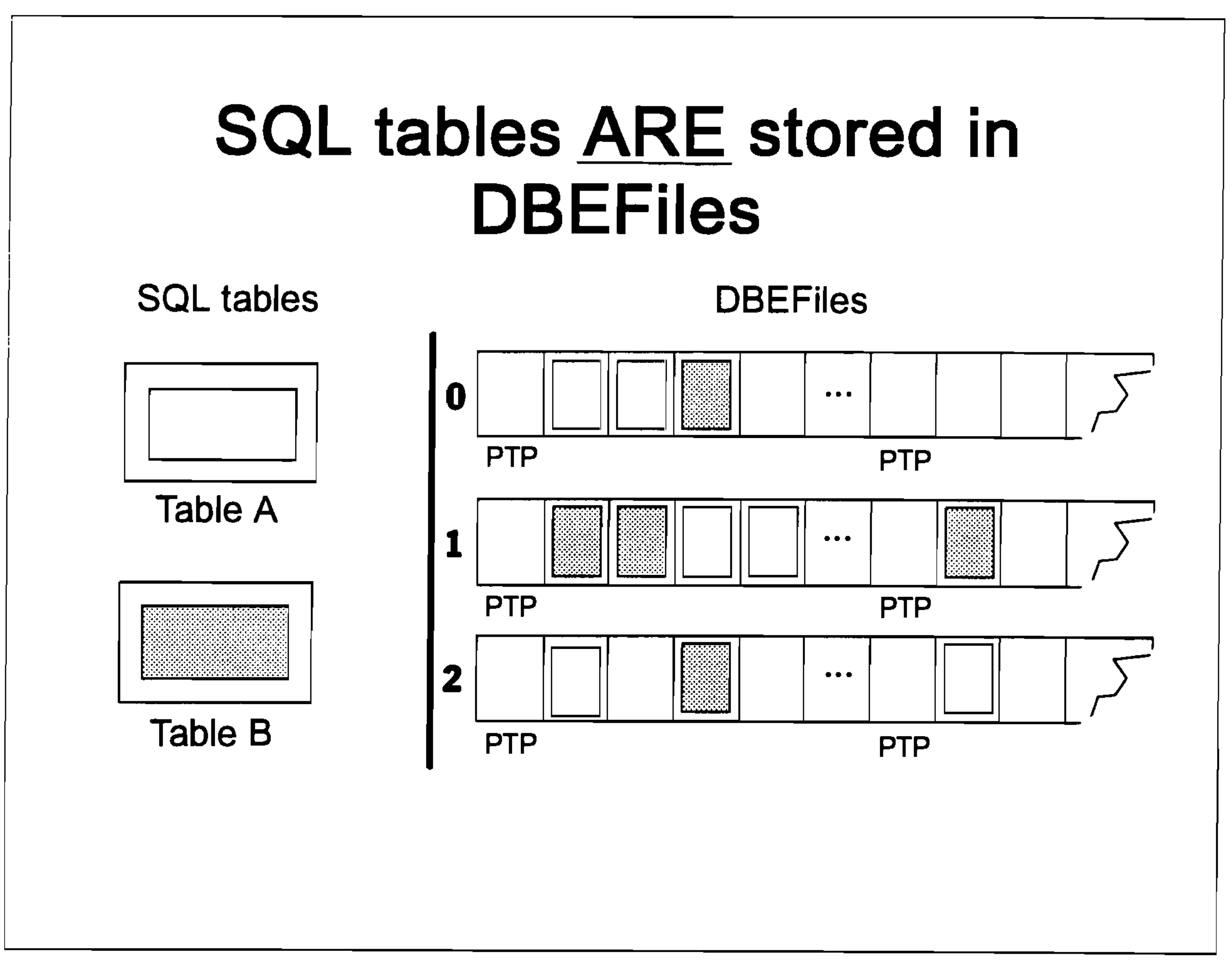
- You can find out which pages your data is stored on by issuing the following query in isql:

```
isql=> select tid(),* from myowner.mytable order by 1;

EXAMPLE
isql=> select tid(),* from recdb.clubs order by 1;
----------------------+--------------+---------+------------------
(TID)                 |CLUBNAME      |CLUBPHONE|ACTIVITY
----------------------+--------------+---------+------------------
                 8:5:0|Energetics    |     1111|       aerobics
                 8:5:1|Windjammers   |     2222|       sailing
                 8:5:2|Downhillers   |     3333|       skiing
                 8:5:3|Poker Faces   |     4444|       cards
                 8:5:4|Spikers       |     5555|       volleyball
                 8:5:5|Stingers      |     6666|       soccer
```

- In Chapter 4, "SQL Transactions and Locking", you will see how DBCORE obtains information out of DBEFiles depending upon the type of scan selected by SQLCORE.

Purpose of Slide

To explain how SQL tables are stored on DBEFiles.



Key Points

- A DBEFile can contain data for many SQL tables.
- Each data page of a DBEFile only contains rows for a single table.

Purpose of Slide

To describe the key elements of security in SQL.

SQL Security Concepts

- WHO
  - A "USER" is a specific logon name (for example, joe@brown).
  - A "GROUP" is a logical entity that ultimately maps to a set of logon names.
- WHAT
  - An "authority" is permission to perform some database task (for example, SELECT authority allows you to read from a table, and INSERT authority allows you to insert new rows).

Key Points

In TurboIMAGE, security is implemented based on how users need to use the database. *User classes* are set up for each unique set of users. Associated with each user class is a number (1 to 63) and a password. The class number is used in the database schema to indicate whether or not the class has read, write or update access to a particular data set or to a subset of data items.

When you initiate access to the database, you supply the password to establish the user class. More than one user can use the same password.

In SQL, security is also implemented based on how users need to use the database, but using a different scheme:

- WHO
  - Normally, each user has a unique *DBEUserID*. On MPE/iX, the format of a DBEUserID is user@account (for example, wolfgang@mrkting). Two or more users would share the same DBEUserID only if the the System Manager allowed the users to use the same user and account information when logging on to the machine.

    On HP-UX, the format of a DBEUserID is the same as the logon name.
  - A *group* is similar to a TurboIMAGE class. Use the CREATE GROUP and ADD USER statements to add users to groups:

    ```
    isql=> CREATE GROUP mygroup;
    isql=> ADD USER wolfgang@mrkting TO GROUP mygroup;
    isql=> ADD USER joe@brown TO GROUP mygroup;
    isql=> COMMIT WORK;
    ```
- WHAT
  - *Authorities* are similar to read and write class lists in TurboIMAGE:
    - *Read access* is similar to SELECT authority in SQL.
    - *Update access* (which also provides read access) is similar to the sum of SELECT and UPDATE authorities in SQL.
    - *Write access* (which also provides read and update access) is similar to the sum of SELECT, UPDATE, INSERT, and DELETE authorities in SQL.

Purpose of Slide

To describe the key elements of security in SQL (continued).

SQL Security Concepts

- HOW
  - Use GRANT statement to give a specific authority to a USER or a GROUP.
  - Use REVOKE statement to take away an authority from a USER or a GROUP.
  - Create views to restrict access to a subset of columns or rows in the table. A view can be thought of as a filter that sits on top of a table.

Key Points

- HOW
  - The GRANT statement gives specified authority to one or more users or authorization groups. You can grant authorities on either tables or views (which are described below).
  - The REVOKE statement takes away authority that was previously granted.
  - In TurboIMAGE, a class can be given access to either an entire data set or a subset of data items from a data set.

    In SQL, a USER or GROUP can be given access to either an entire table or a subset of columns or rows from the table. To give access to a subset, the DBA can create a *view*. Think of a view as a filter that sits on top of the table. The CREATE VIEW statement is used to create a view on a table (or even on a another view).

```
isql => CREATE PUBLIC TABLE Parts.Vendors
> (PartNumber   char(16),
>  VendorNumber INTEGER,
>  VendorCode   INTEGER)
> IN partsfileset;

isql => CREATE VIEW Parts.SpecialVendors
> as SELECT PartNumber,VendorNumber FROM Parts.Vendors
> where VendorCode = 5
> IN partsfileset;

...

isql => select * from Parts.Vendors;           /* THIS IS THE TABLE */
----------------+------------+-----------
PARTNUMBER      |VENDORNUMBER|VENDORCODE
----------------+------------+-----------
1123-P-01       |        9001|          5
1133-P-01       |        9002|          4
1143-P-01       |        9003|          1
1153-P-01       |        9004|          5
1223-MU-01      |        9025|          5
1233-MU-01      |        9006|          4
1243-MU-01      |        9018|          1


isql => select * from Parts.SpecialVendors;   /* THIS IS THE VIEW */
----------------+------------
PARTNUMBER      |VENDORNUMBER
----------------+------------
1123-P-01       |        9001
1153-P-01       |        9004
1223-MU-01      |        9025
```

Purpose of Slide

To provide more information about the GRANT statement.

More about the GRANT command

Syntax: GRANT *Authority [on Table/View]*
TO *{User/Group/PUBLIC}* ;

- The complete syntax is more complicated.
- The following authorities can be granted on a table or view: SELECT, UPDATE, INSERT, and DELETE.
- The following authorities are not granted on a table/view (they are simply granted to a USER or GROUP): CONNECT, DBA, and RESOURCE.

Key Points

- When you grant SELECT, UPDATE, INSERT, or DELETE authority, you must specify a Table or View name.
- **CONNECT** authority is the ability to issue the CONNECT statement (in other words, to simply attach to the DBEnvironment).

  When a normal user initiates access to the DBEnvironment, he or she must issue the CONNECT statement. If the user has not been granted CONNECT authority (or has not been added to a group to which CONNECT authority has been granted), the CONNECT statement will fail and no database access will be possible. SQL CONNECT authority is more restrictive than TurboIMAGE passwords, because it is not possible for an unauthorized user to use a password. However, it is still possible for an unauthorized user to gain entry to a DBEnvironment if the System Manager allows multiple users to share the same user and account logon information at the MPE level.

  Any user or group that has been granted CONNECT authority is part of a special "group" known as PUBLIC. Granting an authority to PUBLIC is an easy way to provide access on same table or view to many users.
- **DBA** authority is the ability to issue any valid SQL statement. A user with DBA authority is exempt from all authorization restrictions. In the SQL world, DBA authority is similar to SM capability on MPE/iX, or being the superuser on HP-UX. The number of users having DBA authority should be small for a production DBEnvironment.
- **RESOURCE** authority is the ability to create tables and authorization groups.
- You cannot grant CONNECT authority to PUBLIC, because PUBLIC is defined as the set of users and groups that have been granted CONNECT authority.

  You also cannot grant DBA or RESOURCE authority to PUBLIC.

Purpose of Slide

To provide an example of how security is implemented in SQL.

SQL Security Example

The DBA issues the following commands:

```
isql => create group Purchasing;
isql => grant connect to Purchasing;
isql => grant select, update, insert, delete
> on PurchDB.OrderItems to Purchasing;
isql => commit work;

isql => add joe@brown to group Purchasing;
isql => add suzy@smith to group Purchasing;
isql => commit work;
```

Now, suzy@smith can do the following:

```
isql => connect to 'PartsDBE';
isql => select * from PurchDB.OrderItems;
```

Key Points

Summary of SQL Security:

- The DBA normally does the following:
  - Creates views on tables.
  - Creates groups to represent each unique set of users.
  - Grants appropriate authorities to each user/group.
  - Adds users (or groups) to groups.
- The users then access the database according to the authorities that have been granted either to them or groups to which they belong.

Purpose of Slide

To explain how SQL security is implemented by IMAGE/SQL.

IMAGESQL and SQL Security

IMAGESQL command

SQL commands issued behind the scene

```
>> SET  TURBODB music
>> SET  SQLDBE   musicdbe
>> ATTACH
          create public table music.albums ...;
          create public table music.composers ...;
          commit work;

          create index albumcode_m1
                on music.albums ...;
          create index composername_m1
                on music.composers ...;
          commit work;
```

- When you issue an IMAGESQL ATTACH command, certain SQL statements are executed behind the scene for you.
  - IMAGESQL essentially issues a CREATE TABLE statement for each data set. By default, the OWNER of the table is the database name. For example:

    ```
    Data set names are albums, composers in the TurboIMAGE database
    named 'music'.  IMAGESQL issues the following SQL statements:

    CREATE PUBLIC TABLE music.albums ...;
    CREATE PUBLIC TABLE music.composers ...;
    ```

  - In the latest release of IMAGE/SQL (B.G1.04 and later), TurboIMAGE keys and third party indexes (TPI) are registered as indexes in the system catalog during the ATTACH command. IMAGESQL essentially issues a CREATE INDEX command for each key/TPI. For example:

    ```
    CREATE INDEX albumcode_m1 ON music.albums ...;
    CREATE INDEX composername_m1 ON music.composers ...;
    ```

Key Points

- Only the TurboIMAGE/XL database creator (DBC) is defined as a user in the DBEnvironment immediately after the IMAGESQL ATTACH. The DBC must add any additional IMAGE/SQL users by using the IMAGESQL ADD USER command (which is explained on the next slide).
- The DBA can view the TurboIMAGE data sets (tables) associated with a database by issuing:

```
isql => select * from system.table where owner='MUSIC';
-------------------+-------------------+--------------------+------+
NAME               |OWNER              |DBEFILESET          |TYPE  |
-------------------+-------------------+--------------------+------+
ALBUMS             |MUSIC              |SYSTEM              |     0|
COMPOSERS          |MUSIC              |SYSTEM              |     0|
LOG                |MUSIC              |SYSTEM              |     0|
SELECTIONS         |MUSIC              |SYSTEM              |     0|
SELECTIONS_A       |MUSIC              |SYSTEM              |     0|
SELECTIONS_A_VO    |MUSIC              |SYSTEM              |     1|
```

- The DBA can view the TurboIMAGE keys associated with a database by issuing:

```
isql => select * from system.imagekey where owner='MUSIC';
-------------------+-------------------+--------------------+------+
INDEXNAME          |TABLENAME          |OWNER               |UNIQUE|
-------------------+-------------------+--------------------+------+
ALBUMCODE_M1       |ALBUMS             |MUSIC               |     1|
COMPOSERNAME_M1    |COMPOSERS          |MUSIC               |     1|
SELECTIONNAME_A1   |SELECTIONS_A       |MUSIC               |     1|
ALBUMCODE_D1       |SELECTIONS         |MUSIC               |     0|
SELECTIONNAME_D2   |SELECTIONS         |MUSIC               |     0|
COMPOSERNAME_D3    |SELECTIONS         |MUSIC               |     0|
ALBUMCODE_D1       |LOG                |MUSIC               |     0|
SELECTIONNAME_D2   |LOG                |MUSIC               |     0|
```

  - The suffix _M1 is used by IMAGESQL when registering manual masters.
  - The suffix _A1 is used by IMAGESQL when registering automatic masters.
  - The suffix _D<n> is used by IMAGESQL when registering search keys on detail data sets (where n can be 1 to 16 depending on the number of keys).
- The DBA can view the third party indexes associated with a database by issuing:

```
isql => select * from system.tpindex where owner='MUSIC';
-------------------+-------------------+--------------------+------+
INDEXNAME          |TABLENAME          |OWNER               |UNIQUE|
-------------------+-------------------+--------------------+------+
```

- This example was generated by using the "Practicing with IMAGE/SQL using MusicDBE" chapter of *Getting Started with HP IMAGE/SQL* (Customer Order Number 36385-90008).

Purpose of Slide

To explain how SQL security is implemented by IMAGE/SQL (continued).

IMAGESQL and SQL Security

IMAGESQL command

SQL commands issued behind the scene

```
>> ADD USER DIR@ACCOUNT WITH PASS=pass20, MODE=1
        create group music_20;
        grant connect to music_20;
        add DIR@ACCOUNT to group music_20;

        create view music.albums_v20 ...;
        create view music.composers_v20 ...;

        grant select,update,insert,delete
              on music.albums_v20 to music_20;
        grant select,update,insert,delete
              on music.composers_v20 to music_20;
        commit work;
```

Key Points

- When you issue an IMAGESQL ADD USER command, certain SQL statements are executed behind the scene for you.
  - If this is the first user to be added for the class/password, then IMAGESQL does the following (in the examples that follow, suppose that an ADD USER is being executed for DIR@ACCOUNT, with a password that maps to class 20):
    - IMAGESQL issues a CREATE GROUP statement for the class/password.

      ```
      CREATE GROUP music_20;
      ```

    - IMAGESQL issues a GRANT statement so that anyone in the group can connect to the DBEnvironment, and also an ADD TO GROUP statement to put DIR@ACCOUNT into the new group.

      ```
      GRANT CONNECT TO music_20;
      ADD DIR@ACCOUNT TO GROUP music_20;
      ```

    - IMAGESQL issues a CREATE VIEW statement for each data set that the user class is allowed to access, according to the TurboIMAGE schema. Only the data items that the class is authorized to access are visible in the view. The name of the view has a '_V' and the class number appended at the end.

      ```
      CREATE VIEW music.albums_v20 ...;
      CREATE VIEW music.composers_v20 ...;
      ```

    - Finally, IMAGESQL performs a GRANT on the new view, so that anyone in the group has the appropriate authorities on the view. In our example, class 20 has write access (also known as full data access) according to the TurboIMAGE schema. Write access maps to SELECT, UPDATE, INSERT, and DELETE authorities in SQL:

      ```
      GRANT SELECT,UPDATE,INSERT,DELETE on music.albums_v20
         TO music_20;
      GRANT SELECT,UPDATE,INSERT,DELETE on music.composers_v20
         TO music_20;
      ```

  - If this is the second (or other) user for the class/password, then IMAGESQL simply issues an ADD TO GROUP statement to add the new user to the appropriate group for the class/password.

    ```
    IMAGESQL >> ADD USER PINKY@ACCOUNT WITH PASS=pass20, MODE=1

         SQL => ADD PINKY@ACCOUNT TO GROUP music_20;
    ```

  - When users in the group want to access information in the TurboIMAGE data set, they must use the view that has been set up for them:

    ```
    isql => select * from music.albums_v20;
    ```

3

Trouble-Shooting with SQLMON

Purpose of Slide

To explain how to start SQLMON.

Starting SQLMON

```
:sqlmon

Welcome to SQLMONITOR!  Type HELP MAIN for more information.

SQLMONITOR SUBSYSTEMS (and abbreviations):
     OVERVIEW   IO   LOAD   LOCK   SAMPLEIO   STATIC
        /o      /i   /loa   /loc     /sa        /st

CURRENT SUBSYSTEM SCREENS:
          OVERVIEW   SESSION   PROGRAM
              o         s         p

SQLMONITOR  OVERVIEW  => help main
SQLMONITOR  HELP OVERVIEW  => //
SQLMONITOR  OVERVIEW  => set dbenv 'musicdbe'
```

Key Points

SQLMON is an online diagnostic tool that monitors the activity of a DBEnvironment. SQLMON screens provide information on file capacity, locking, I/O, logging, tables, and indexes. They summarize activity for the entire DBEnvironment, or focus on individual sessions, programs, or database components. SQLMON is a read-only utility, and cannot modify any aspect of the DBEnvironment.

- To run SQLMON, simply type *sqlmon* at the system prompt.
  - On MPE/iX, SQLMON consists of two files:
    - SQLMONP.PUB.SYS is the executable program.
    - SQLMON.PUB.SYS is a command file which issues file equations and a command to "run sqlmonp.pub.sys".

    When you type *sqlmon* to invoke the program, you are actually executing the command file.
  - On HP-UX, the executable is stored under /usr/bin/sqlmon. When you type *sqlmon* to invoke the program, you are actually executing the program file.
- If you are a new user, type ***help main*** to learn more about SQLMON basics (such as the SET DBENVIRONMENT command, and how to navigate the SQLMON screens).
- Issue the SET DBENVIRONMENT command to 'connect' to the DBEnvironment. To successfully issue this command, one of the following must be true: 1) you are the DBECreator, 2) you have system manager capability, or 3) you know the DBEnvironment maintenance word.

```
SQLMONITOR  OVERVIEW  => set dbenv musicdbe [ MAINT=maint ]

DBEnvironment is not active (no other session in progress).  (DBWARN 34505)

At least one TurboIMAGE database is attached to this DBEnvironment (DBWARN
34526).  TurboIMAGE locks are not visible in SQLMON.  Use DBUTIL to
determine locks granted on TurboIMAGE objects.
```

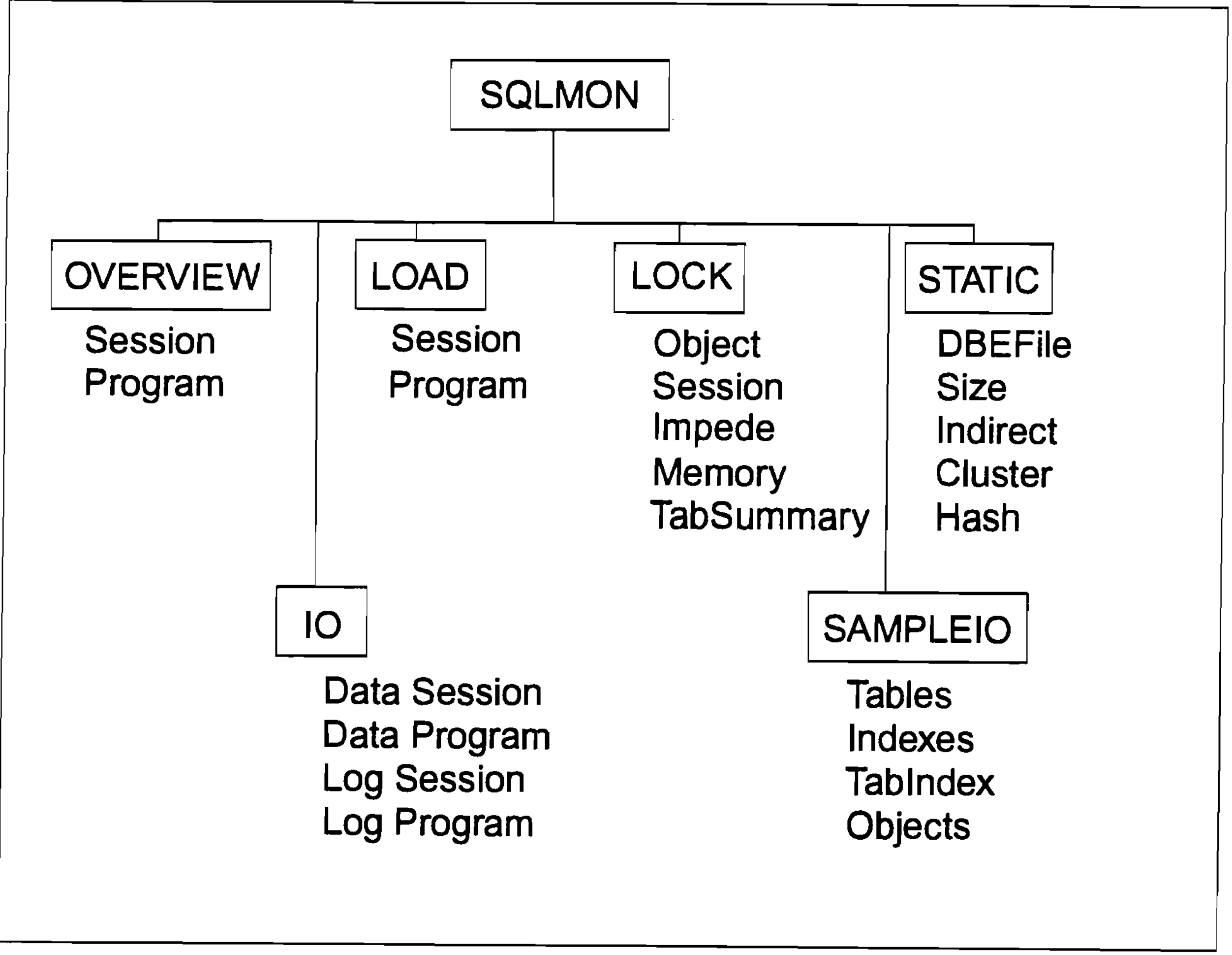
  The MAINT word is not required if you are the DBECreator or you have system manager capability.

  If nobody is connected to the DBE, then a warning is issued. Many SQLMON screens cannot be accessed if the DBE is not active.

  A warning is issued when an IMAGE/SQL DBE is accessed, to remind you that SQLMON only shows locks on ALLBASE/SQL tables (including the system catalog). You must use DBUTIL to see locks on TurboIMAGE objects.
- For more information about SQLMON, see Chapters 6 - 9 of the *ALLBASE/SQL Performance and Monitoring Guidelines* document (HP Part No. 36216-90102 on MPE/iX and 36217-90185 on HP-UX).

Purpose of Slide

To explain how the SQLMON screens and subsystems are organized.



- SQLMON screens are organized into subsystems. Each subsystem is a set of screens that are logically related to each other.
  - **OVERVIEW** shows general performance information.
  - **IO** provides information on data and log buffer activity.
  - **LOAD** displays transaction throughput data.
  - **LOCK** shows locking activity.
  - **SAMPLEIO** provides information on DBEFile, table and index I/O.
  - **STATIC** provides relatively non-dynamic information about DBEFiles, tables, and indexes (such as current size, length of hash overflow chains, etc.)

  Each subsystem consists of a main screen, which has the same name as the subsystem, and additional "detail" screens. For example the screens in the OVERVIEW subsystem are:
  - OVERVIEW (main screen),
  - OVERVIEW SESSION (detail screen)
  - OVERVIEW PROGRAM (detail screen)

Key Points

- The main screen of a subsystem contains a high-level or summary information for the subsystem, and the other screens give more detailed information. Most of the time, the user will first visit the main screen of a subsystem to obtain the "big picture", and afterwards will visit detail screens to obtain more specific information of interest.
- When you access a screen, you automatically move to the subsystem that the screen belongs to. The name of the current subsystem is displayed in the prompt. For example:

```
SQLMONITOR  OVERVIEW  =>          /* you are in the OVERVIEW subsystem. */

SQLMONITOR  LOCK  =>              /* you are in the LOCK subsystem.     */
```

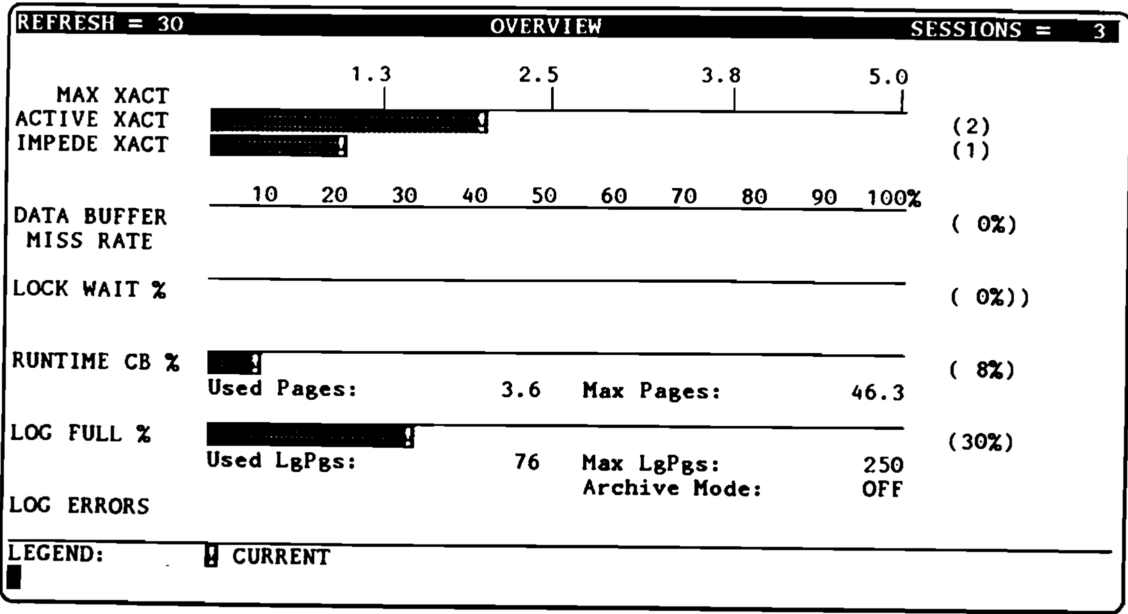
- You access a screen by typing the name of the screen (or an abbreviation) at the prompt. A menu is printed to help you remember the names of screens (and abbreviations) that you should type.

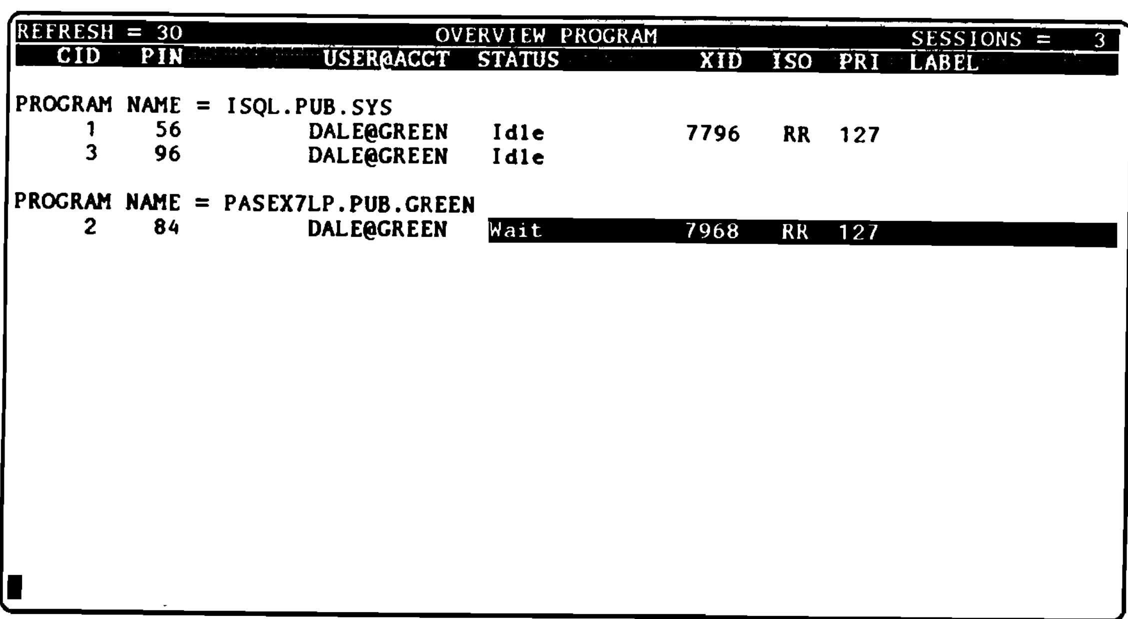
```
SQLMONITOR SUBSYSTEMS (and abbreviations):

           OVERVIEW    IO    LOAD    LOCK    SAMPLEIO    STATIC
              /o       /i    /loa    /loc       /sa        /st

CURRENT SUBSYSTEM SCREENS:

                         OVERVIEW     SESSION     PROGRAM
                            o            s           p

/***************************************************************************/
/* To obtain a screen in the current subsystem ...                         */
/***************************************************************************/

SQLMONITOR  OVERVIEW  => o     /* To obtain the OVERVIEW screen          */

SQLMONITOR  OVERVIEW  => s     /* To obtain the OVERVIEW SESSION screen */

SQLMONITOR  OVERVIEW  => p     /* To obtain the OVERVIEW PROGRAM screen */

/***************************************************************************/
/* To obtain a screen in another subsystem ...                             */
/***************************************************************************/

SQLMONITOR  LOCK  => /o        /* To obtain the OVERVIEW screen          */

SQLMONITOR  LOCK  => /o s      /* To obtain the OVERVIEW SESSION screen */

SQLMONITOR  LOCK  => /o p      /* To obtain the OVERVIEW PROGRAM screen */
```

- The OVERVIEW, LOCK, and STATIC subsystems are perhaps the most useful to the typical IMAGE/SQL DBA, so I'll briefly cover these on the next few pages.

Purpose of Slide

To briefly describe the screens in the OVERVIEW subsystem.





Key Points

- The DBEnvironment must be active to obtain any screen in the subsystem (in other words, at least one user must be connected to the DBE). A warning is returned if you try to access one of these screens if nobody is connected.

OVERVIEW Screen

This screen provides an overall view of some of the most interesting aspects of the performance of the DBEnvironment, including the data buffer pool miss rate, the current size of the runtime control block, and the current size of the log file. Some information on this screen is examined in greater detail on the LOAD and IO screens.

OVERVIEW SESSION Screen

This screen identifies all sessions connected to the DBEnvironment.

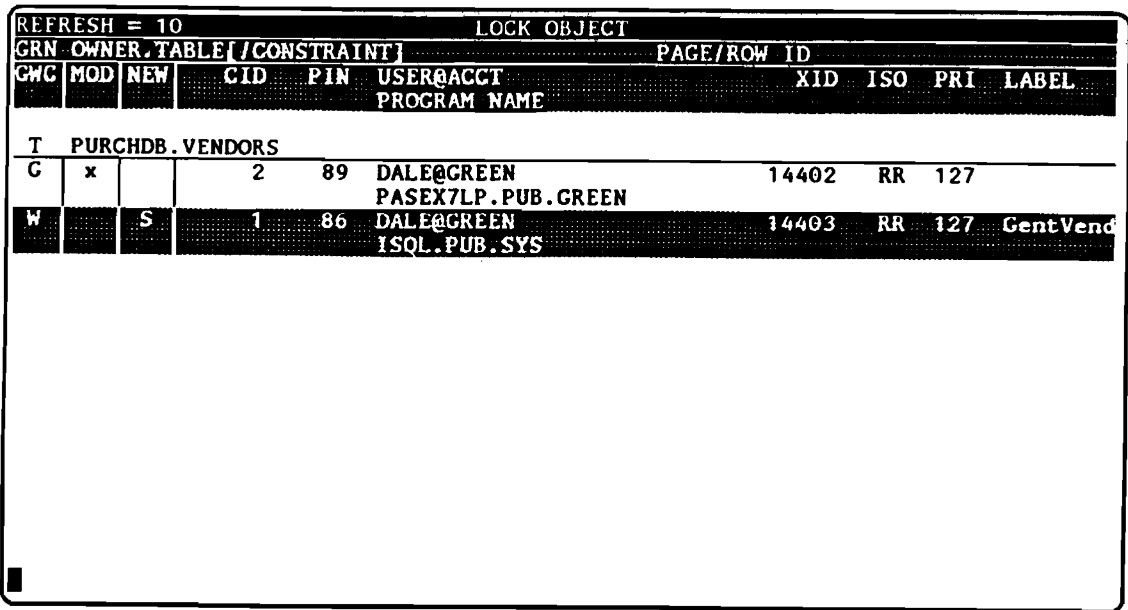
OVERVIEW PROGRAM Screen

This screen groups together all sessions running the same program. This information can help you determine if a performance problem is related to a particular program, as opposed to simply related to a particular session.

Purpose of Slide

To briefly describe the screens in the LOCK subsystem.

```
REFRESH = 10              LOCK                    LOCKFILTER = SU/TPR/GWC/SXRsxr6v/1
G  OWNER.TABLE[/CONSTRAINT]                       PAGE/ROW ID|       LOCK QUEUE
T  DBCORE.MARSCH                                             |s
T  DBCORE.MARSCS                                             |ss
T  DBCORE.MARSINDX                                           |ss
T  DBCORE.MARSREL                                            |ss
T  HPRDBSS.COLUMN                                            |s
T  HPRDBSS.DBEFILESET                                        |s
T  HPRDBSS.INDEX                                             |s
T  HPRDBSS.SECTION                                           |s
T  HPRDBSS.SPECAUTH                                          |s
T  HPRDBSS.TABAUTH                                           |s
T  HPRDBSS.TABLE                                             |s
T  PURCHDB.VENDORS                                           |Sx
T  STOREDSECT.SYSTEM                                         |s

P  DBCORE.MARSCH                                  0:0:2:0    |s
P  DBCORE.MARSCS                                  0:0:3:0    |ss
P  DBCORE.MARSINDX                                0:0:5:0    |ss
P  DBCORE.MARSREL                                 0:0:1:0    |ss
P  HPRDBSS.COLUMN                                 0:0:103:0  |S
P  HPRDBSS.DBEFILESET                             0:0:26:0   |S
P  HPRDBSS.INDEX                                  0:0:98:0   |S
Continue? ([YES],NO)
```



Key Points

- The DBEnvironment must be active to obtain any screen in the subsystem (in other words, at least one user must be connected to the DBE). A warning is returned if you try to access one of these screens if nobody is connected.

LOCK Screen

This screen provides information about lock activity for the entire DBEnvironment.

The most important use of this screen is to identify the lock objects that have waiters or converters. These lock objects are the bottlenecks in your DBEnvironment which cause one or more sessions to wait before they may complete their processing. Every effort should be made to tune your DBEnvironment to minimize the amount of waiting that sessions must perform. You can issue the SET LOCKFILTER command to restrict the display of locks on this screen, so that only lock objects that are causing contention are shown. Afterward, you can use the LOCK OBJECT screen to identify the sessions in the queue for each particular lock.

LOCK OBJECT Screen

This screen provides information about all of the sessions that have been granted access to a single lock object, are waiting for access to the lock object, or are converting an existing lock on the lock object to a stronger one.

LOCK SESSION Screen

This screen provides lock activity information for a single session.

The LOCK SESSION screen answers the question "What locks have been granted to this session, and what locks is it waiting to acquire?".

LOCK IMPEDE Screen

This screen identifies sessions that are waiting for locks granted to a particular session.

The LOCK IMPEDE screen answers the question "What locks have been granted to this session that are causing other sessions to wait?"

LOCK MEMORY Screen

This screen allows you to identify how many locks are allocated to each session.

LOCK TABSUMMARY Screen

This screen allows you to identify how many locks are allocated at each granularity for each table. The screen can either be used to summarize the locks that have been allocated for use by a particular session, or to summarize all locks that exist in the DBEnvironment.

Purpose of Slide

To briefly describe the screens in the STATIC subsystem.

```
PartsDBE                              STATIC
DBEFILESET
    HASH?  IMAGE?  NUMIDX  TYPE          OWNER.TABLE

FILEFS
                      0    PUBLIC        STOREDSECT.FILEFS
                      0    PUBLIC        PURCHDB.REPORTS(3)

INVOICE3FS
                      0    PUBLIC        STOREDSECT.INVOICE3FS

INVOICEFS
                      0    PUBLIC        STOREDSECT.INVOICEFS
                      1    PUBLIC        PURCHDB.CUSTOMER
                      0    PUBLICROW     PURCHDB.INVOICE
          *           0    PUBLIC        PURCHDB.SALESDATA

ORDERFS
                      0    PUBLIC        STOREDSECT.ORDERFS
                      1    PUBLIC        PURCHDB.ORDERITEMS
                      2    PUBLIC        PURCHDB.ORDERS
                      0    PUBLIC        PURCHDB.REPORTS

Continue? ([YES],NO)
```

```
PartsDBE                              STATIC DBEFILE
DBEFILESET                   DBEFILESET  FULLNESS     %   FSUSED PAGES  FSMAX PAGES
            DBEFILE TYP BD   DBEFILE     FULLNESS     %     USED PAGES    MAX PAGES

FILEFS                                               4%         2             50
          FILEDATA  TBL                              4%         2             50

INVOICE3FS                                           5%         1             22
     INVOICE3DATAF1 MIX                              5%         1             22

INVOICEFS                                           84%        47             56
      INVOICEDATAF1 MIX                             92%        11             12
      INVOICEDATAF2 TBL                             64%        14             22
      INVOICEDATAF4 MIX  *                         100%        22             22

ORDERFS                                DETACHED                              100
        ORDERDATAF1 TBL                DETACHED                               50
        ORDERINDXF1 IDX                DETACHED                               50

PURCHFS                                              8%         8            100
        PURCHDATAF1 TBL                              6%         3             50
        PURCHINDXF1 IDX                             10%         5             50

Continue? ([YES],NO)
```

Key Points

- The DBEnvironment does not need to be active to obtain any screen in the subsystem.

STATIC Screen

This screen provides miscellaneous information about each DBEFileSet within the DBEnvironment, including the names of all tables in the DBEFileSet, the number of B-tree indexes and referential constraints (PCRs) that are defined on each table, and whether or not a table is actually a TurboIMAGE data set.

STATIC DBEFILE Screen

This screen provides information about the capacity and fullness of each DBEFileSet existing in the DBEnvironment. The capacity and fullness of each DBEFile contained within the DBEFileSet(s) is also provided. This information can be used to determine whether or not space should be added to or removed from the DBEnvironment.

STATIC SIZE Screen

This screen provides information about the size of tables, B-tree indexes, and referential constraints (PCRs) contained within a DBEFileSet.

STATIC INDIRECT Screen

This screen provides information about the percentage of indirect rows that exist in each table of a DBEFileSet. An indirect row is one that can only be accessed by first fetching one page to obtain the address of the row, and then fetching a second page to actually obtain the row data. Indirect rows increase the amount of I/O that must be performed to obtain data.

STATIC CLUSTER Screen

This screen provides information about the clustering of B-tree indexes and referential constraints (PCRs) contained within a DBEFileSet. Applications which frequently access data in index order (which includes using an ORDER BY, GROUP BY, DISTINCT, or UNION clause) will have better performance if the table data is "clustered" (physically stored on disk in index order). Performance is improved because I/O is minimized.

STATIC HASH Screen

The STATIC HASH screen provides information about the overflow chains associated with hashed tables.

4

SQL Transactions and Locking

Purpose of Slide

To describe how transactions are used.

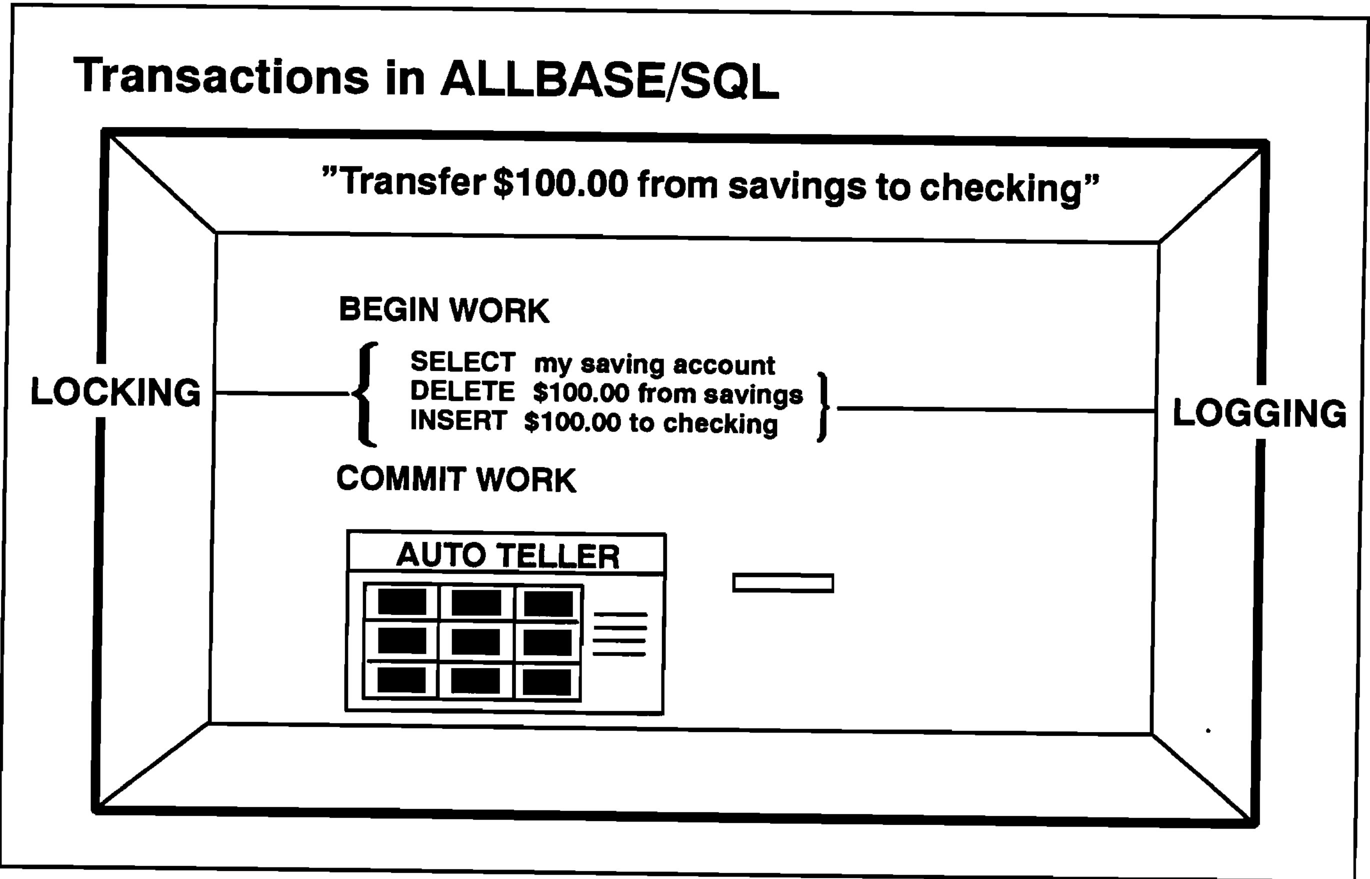


Figure 4-1.

Key Points

- A transaction consists of one or more SQL statements. A transaction begins with a `BEGIN WORK` statement and ends with either a `COMMIT WORK` or a `ROLLBACK WORK` statement.
- A transaction is a unit of work. Either all of the statements contained within a transaction will be executed, or none of them will.
- Locks are normally obtained by transactions when SQL statements are executed. By default, locks are held until the transaction ends.
- Locking can degrade performance in two ways:
  - Locking reduces concurrency—**Concurrency** is the degree to which data can be simultaneously accessed by multiple users. For example, a table that can be accessed by one hundred users at a time has better concurrency than a table that can only be accessed by one user at a time. Locking affects the number of users that can access a specific portion of data at the same time. For example, if one user is updating a row, no other user is allowed to access that row until the first user is done. A transaction must wait if the data that it needs is already locked in an incompatible mode by another transaction; when transactions wait, concurrency is reduced.
  - Deadlocks sometimes occur—A **deadlock** occurs when two or more transactions are waiting for each other to complete. A deadlock degrades performance because CPU must be used by:
    - ALLBASE/SQL to resolve the deadlock situation (one of the transactions is automatically rolled back).
    - The application program to redrive the cancelled transaction.
  - Deadlocks will be explained in greater detail later in this module.
- The major objectives of transaction management are:
  - Ensure logical data integrity. Transactions should be written so logical data corruption does not occur. For example, a transaction should not update a row without locking it first; when the row is locked, its value cannot change before the update completes. If the row is not locked and its value changes before the update completes, the updating transaction will overwrite (that is, lose) the changes made by another transaction.
  - Minimize lock contention (maximize concurrency and minimize deadlocks).

Purpose of Slide

To explain how to build transactions.

Rules For Building Transactions

1. A transaction is a unit of work
2. Keep it short
3. Keep "extra" processing out of it
4. Retrieve user input before it starts

Figure 4-2.

Key Points

- When building transactions, keep the following in mind:
  - A transaction is a unit of work.
    - Keep operations that are required to maintain logical data integrity within a single transaction. If a row should be added to Table2 only if a specific row exists in Table1, then `SELECT` on Table1 and `INSERT` into Table2 in the same transaction. This may reduce concurrency, but is needed to prevent data corruption.
  - Keep transactions short.
    - Make a transaction only as long as it needs to be to perform its specific function. Each additional SQL statement generally acquires more locks. Locks are normally held until the transaction commits. The longer the transaction, the greater the potential that the transaction will hold a lock that might be needed by another transaction.
  - Keep non-database processing outside of transactions.
    - Extra processing takes time, so locks might be held longer than they need to be.
  - Retrieve all user input before the start of a transaction.
    - Keep terminal reads and writes (especially user prompts) outside of transactions to ensure that locks are not held when someone walks away from the terminal.

Purpose of Slide

To describe how to start a transaction.

Starting Transactions

DBA Sets :	Users Issue :
MAX Transactions = 3	User 1: BEGIN WORK => active XACT
	User 2: BEGIN WORK => active XACT
	User 3: BEGIN WORK => active XACT
	User 4: BEGIN WORK => WAIT

Figure 4-3.

Key Points

- Transactions are started with a **BEGIN WORK** statement, but the command does not need to be explicitly issued by the user. ALLBASE/SQL will automatically issue a **BEGIN WORK** statement if another SQL statement is executed and a transaction has not already been started.
- Issuing explicit **BEGIN WORK** statements is good programming practice, and is required if you wish to specify a value other than the default for the isolation level or the priority of the transaction (these attributes will be explained later in this module).
- A transaction that has been started but has not been terminated is known as an *active* transaction.
- ALLBASE/SQL assigns a unique id for each active transaction.
- SYSTEM.TRANSACTION is a pseudo-table that displays all active transactions in the DBEnvironment. It includes the name of the user who started the transaction, the transaction id, and the priority of the transaction.
- **Max Transactions** is a parameter in the DBECON file that limits the number of concurrent, active transactions in the DBEnvironment. This parameter can be specified in the **START DBE** statement, or by using the SQLUTIL **ALTDBE** command. The default value is 2. Two times the number of concurrent users is a good setting for Max Transactions.
- A transaction that is started after the transaction limit has been reached is placed onto a wait queue called the throttle wait list, and is known as a **throttled transaction**. A throttled transaction must wait until one of the active transactions terminates or its own timeout limit is reached. All throttled transactions in the DBEnvironment can be identifed by issuing the following query from ISQL:

```
SELECT * FROM SYSTEM.CALL WHERE STATUS = 'Waiting - SERVER';
```

Note The case is very important in the above command. Enter it exactly as shown.

Purpose of Slide

To describe how to use transaction timeout limits.

Transaction Timeout Limits

DBA Sets :	User issues :
Maximum Timeout = 10 hours	Set user timeout to MAXIMUM;
	or
	Set user timeout to DEFAULT;
Default Timeout = 1 hour	or
	Set user timeout to 5 minutes;

Figure 4-4.

Key Points

- The timeout limit for a transaction is controlled by the **SET USER TIMEOUT** statement, and the MaximumTimeout and DefaultTimeout parameters in the DBECON file. The **SET USER TIMEOUT** statement specifies the amount of time a transaction will wait if a required database resource (such as a transaction slot, or a lock) is not available. You can set the timeout limit to MAXIMUM (in which case the MaximumTimeout value is used), DEFAULT (the DefaultTimeout is used), or to an explicit number of seconds or minutes. If an explicit number is specified, the value must be less than or equal to MaximumTimeout, or the statement will fail.
- If you do not issue a **SET USER TIMEOUT** statement before you start a transaction, the timeout limit for the transaction is automatically set to the DefaultTimeout value.
- The default value for MaximumTimeout is NONE, which means that a transaction will never timeout (that is, it will wait as long as is necessary to obtain a database resource). The default value for DefaultTimeout is MAXIMUM, which means that the MaximumTimeout is used as the default. These DBECON parameters can be modified in the **START DBE** statement, or by using the SQLUTIL **ALTDBE** command.

Purpose of Slide

To describe how to terminate a transaction.

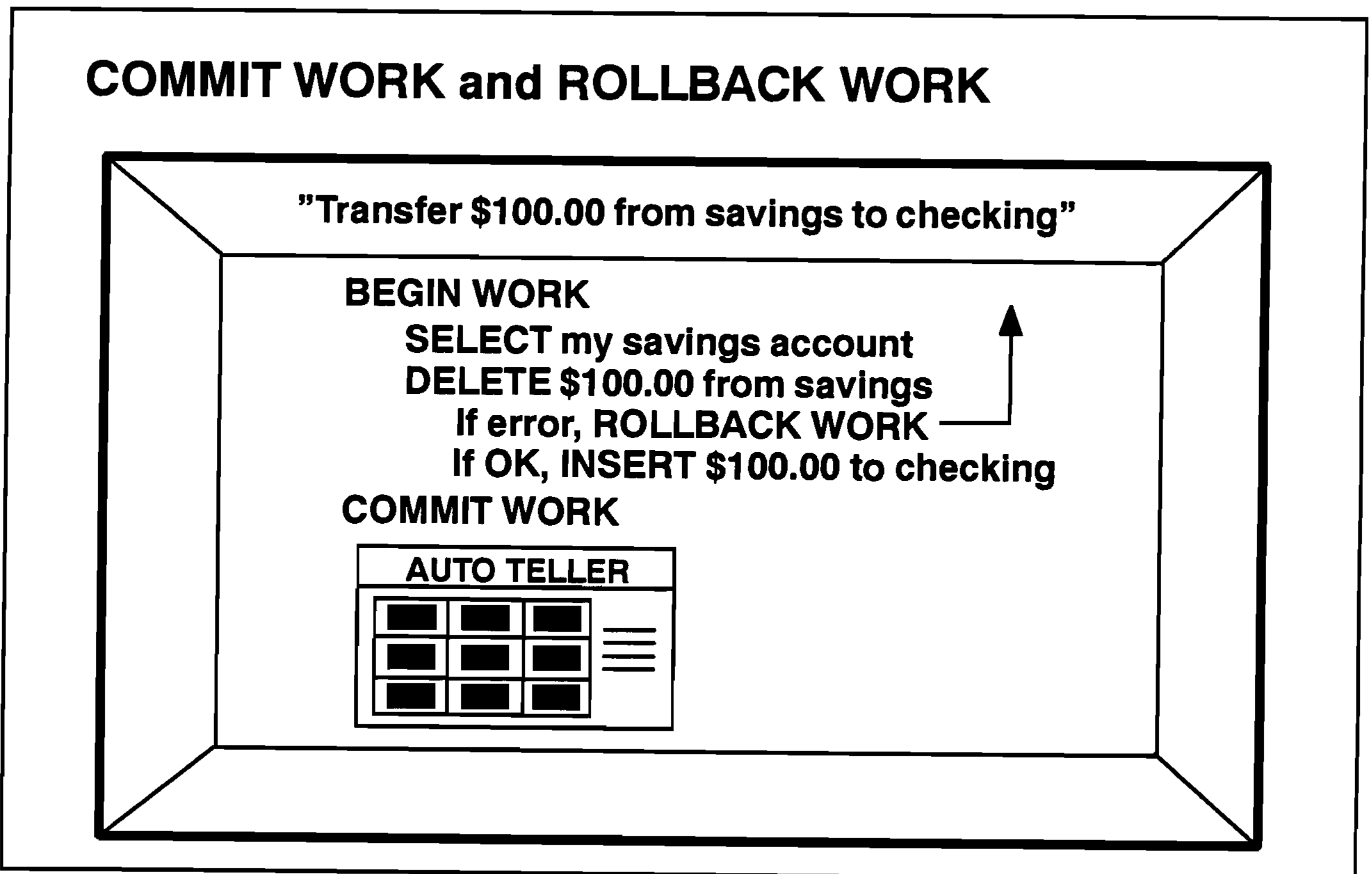


Figure 4-5.

Key Points

- Transactions are ended with either a `COMMIT WORK` or a `ROLLBACK WORK` statement. All locks are released when a transaction is ended.
  - `COMMIT WORK` makes all data modifications made by the transaction permanent.
  - `ROLLBACK WORK` undoes all operations since the `BEGIN WORK` statement was issued. The `ROLLBACK WORK` statement can be useful under the following situations:
    - The SQLCA.SQLCODE indicates an error has occurred. (SQLCA is a special record variable that is updated by ALLBASE/SQL during every SQL call. It can be used to programmatically detect whether or not a call executes successfully.)
    - `UPDATE`, `INSERT`, or `DELETE` statements are issued on multiple rows. It is possible that one of the commands might fail after only *some* of the target rows had been operated on, and you want to undo the changes that were made (restore your data back to a consistent state).
    - You provide input indicating that you do not wish to commit the transaction.
- Only the transaction of the user that issues the `COMMIT WORK` or the `ROLLBACK WORK` is affected by the statement.
- Transactions can also be rolled back by ALLBASE/SQL without the user explicitly issuing the `ROLLBACK WORK` statement:
- If you RELEASE from a DBEnvironment without issuing a `COMMIT WORK`, ALLBASE/SQL automatically rolls back the transaction.
- If you have an active transaction in ISQL when you issue an `EXIT` command, you will be prompted about whether or not you want to commit the transaction. ISQL is being friendly by remembering that a transaction is in progress and asking you how to terminate it before it implicity issues a `RELEASE` statement. But if you issue a `RELEASE` statement explicity in ISQL while you have an active transaction, you will receive a message indicating that your transaction was aborted.
- When a soft-crash or a hard-crash occurs, ALLBASE/SQL automatically rolls back all transactions that were active at the time of the crash.
- When a deadlock occurs, ALLBASE/SQL automatically rolls back one of the transactions involved in the deadlock.

Purpose of Slide

To describe a deadlock and how it is resolved by ALLBASE/SQL.

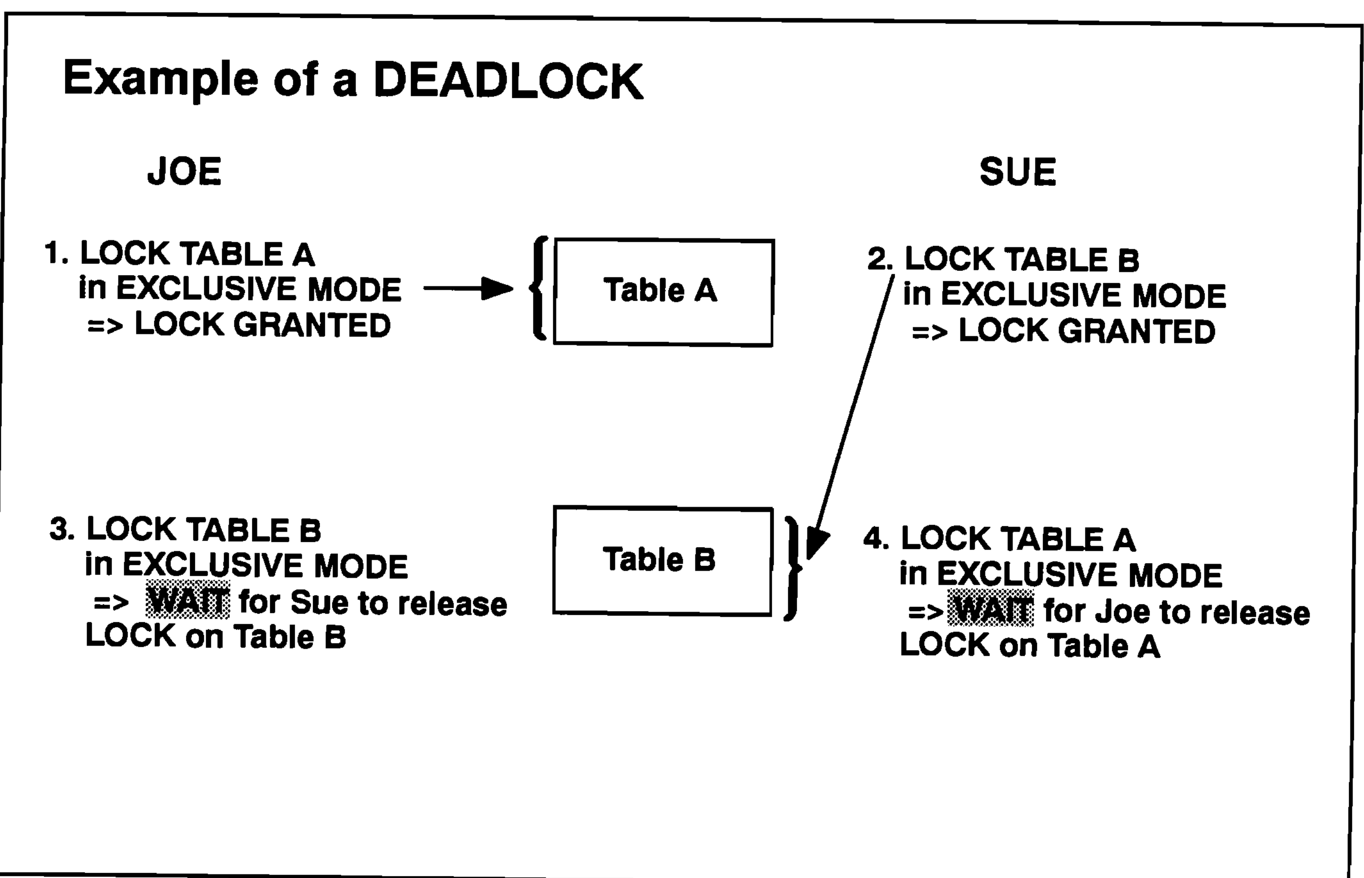


Figure 4-6.

Key Points

- A deadlock occurs when at least two transactions are waiting for each other to complete. Each has already obtained a lock on data that the other needs to complete its work. Each would wait forever to obtain the other's lock if ALLBASE/SQL did not detect the deadlock situation and issue a `ROLLBACK WORK` for one of the transactions. Deadlocks are not always this simple. Sometimes a ring of transactions enter into a deadlock.
- The **priority** of a transaction is an integer value from 0 to 255 that can be specified during the `BEGIN WORK` statement. A priority of 127 is assigned by default if no value is explicitly specified.
- If only two transactions are involved in the deadlock, the transaction having the largest priority number is rolled back to resolve the deadlock. If both transactions have the same priority, then the most recent transaction is rolled back.
- If more than two transactions are involved in the deadlock, ALLBASE/SQL resolves the deadlock by choosing a victim from only two of the transactions: the last transaction to enter the ring (the one that caused the deadlock loop to be closed), and the transaction that is waiting for it. The victim is selected using the rules described above on these two transactions. Notice that the transaction that is aborted may not be the transaction with the largest priority number or the most recent transaction among all transactions that are involved in the deadlock.

Purpose of Slide

To describe savepoints.

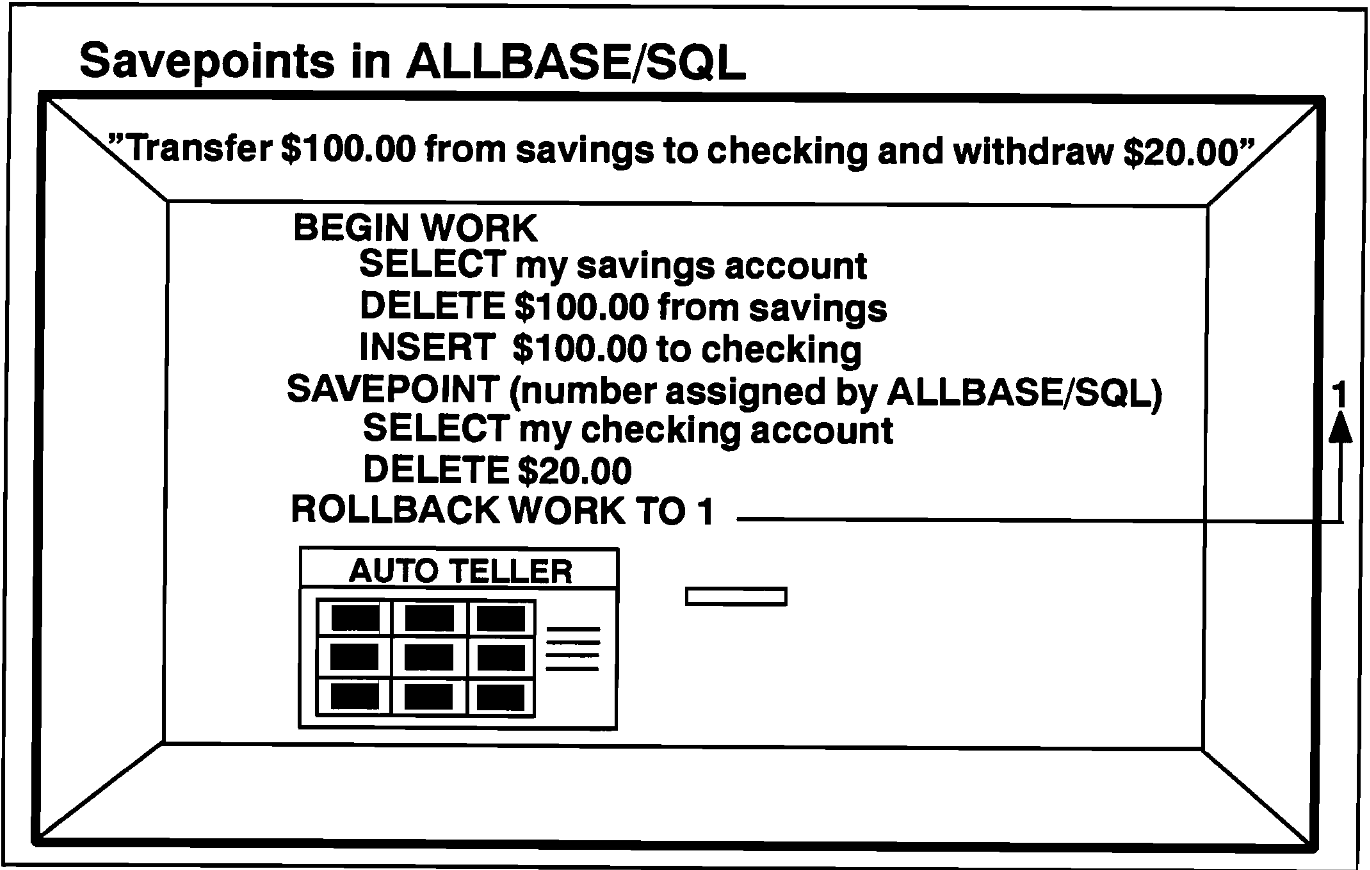


Figure 4-7.

Key Points

- A SAVEPOINT is a transaction marker that allows you to roll back part of a transaction. ROLLBACK TO SAVEPOINT will not end the transaction or release locks that were obtained prior to the setting of the SAVEPOINT, but it will release locks that were obtained after the SAVEPOINT was issued.
- A SAVEPOINT can be placed anywhere between the BEGIN WORK and COMMIT WORK statements.
- Multiple SAVEPOINT statements can be issued within a transaction. ALLBASE/SQL assigns a number to each one. The number is returned programmatically if a host variable is provided when the SAVEPOINT statement is issued.
- The first SAVEPOINT number returned in a transaction is 1. The largest possible SAVEPOINT number is (2**31)-1.
- In order to ROLLBACK TO SAVEPOINT, the number of the appropriate SAVEPOINT must be specified in the ROLLBACK WORK statement. Please refer to the example for ROLLBACK WORK on the previous page.
- The slide shows that the SAVEPOINT is useful when the user decides not to withdraw the $20.00, but still wishes the $100.00 transferred.

Purpose of Slide

To describe how ALLBASE/SQL uses locking.

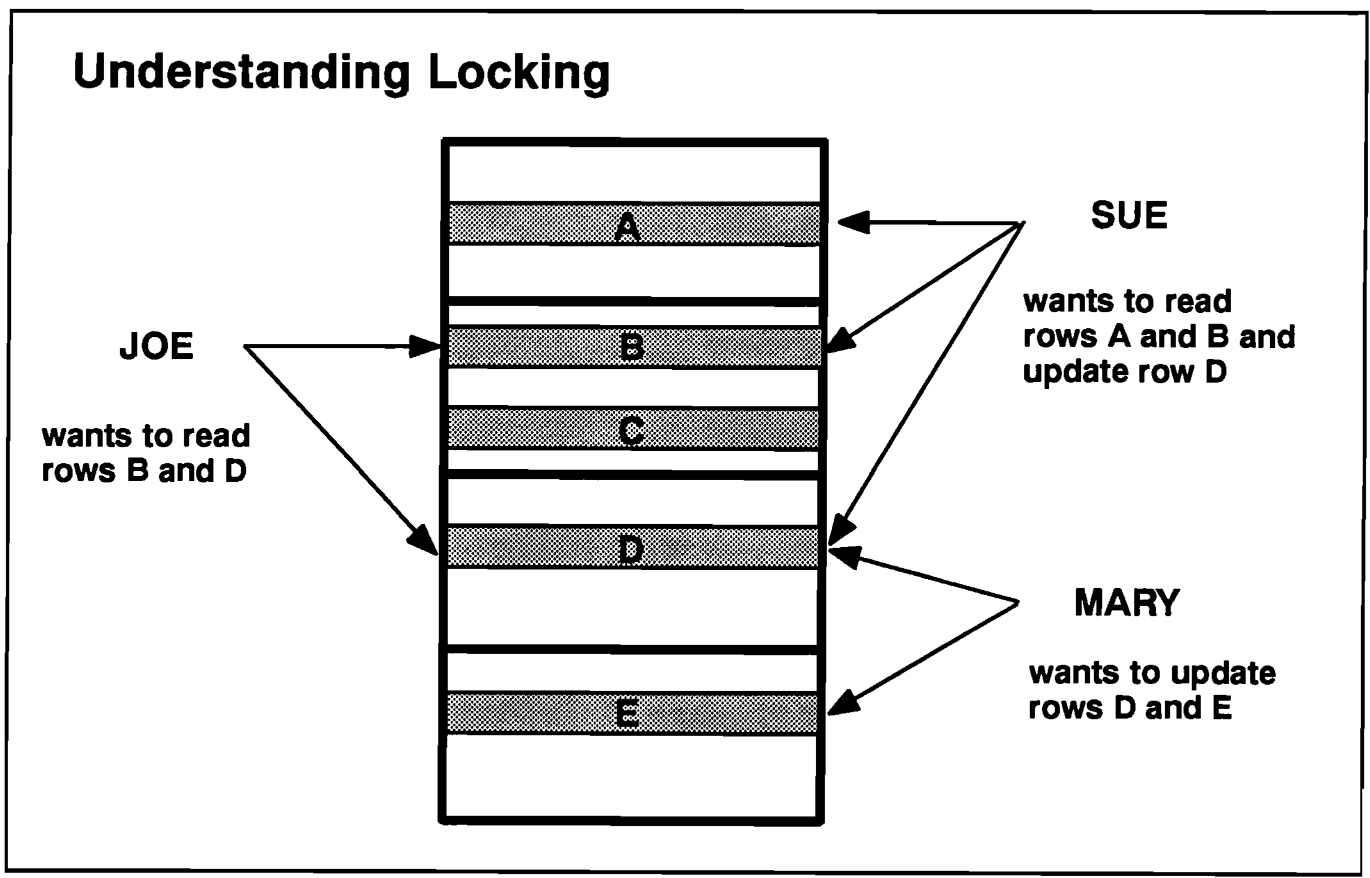


Figure 4-8.

Key Points

- ALLBASE/SQL uses locks to regulate concurrent access to the same data.
- Locking is needed to ensure data integrity in a multi-user environment. Without appropriate locking, you might incorrectly overwrite changes made by other users, or read uncommitted data.
- Locking can degrade performance in two ways:
  - Locking reduces concurrency.
  - Deadlocks sometimes occur.
- A well tuned application has a high rate of concurrency and a low rate of deadlock. In reality, however, a tradeoff is usually necessary. A low rate of deadlock is often achieved by locking more of the data than you actually need (for example, locking the entire table instead of most of the pages in the table). This strategy tends to increase the lock wait time for other transactions. Conversely, a short wait time for locks is usually achieved by locking small portions of the data. This strategy can increase the number of deadlocks.
- ALLBASE/SQL supports a variety of lock granularities, lock types, table types, and isolation levels to enable a transaction to lock only what is necessary to keep other transactions from interfering with its work (these concepts will be explained in this module). ALLBASE/SQL application developers can use these features to develop programs that maximize concurrency and minimize deadlocks.
- Remember that locks are released when a transaction terminates. It is important that all transactions are terminated properly. All of the following SQL statements can be used to terminate transactions: `COMMIT WORK`, `ROLLBACK WORK`, `RELEASE`, `STOP DBE`, and `TERMINATE USER`.

Purpose of Slide

To identify DBECon file parameters associated with locking.

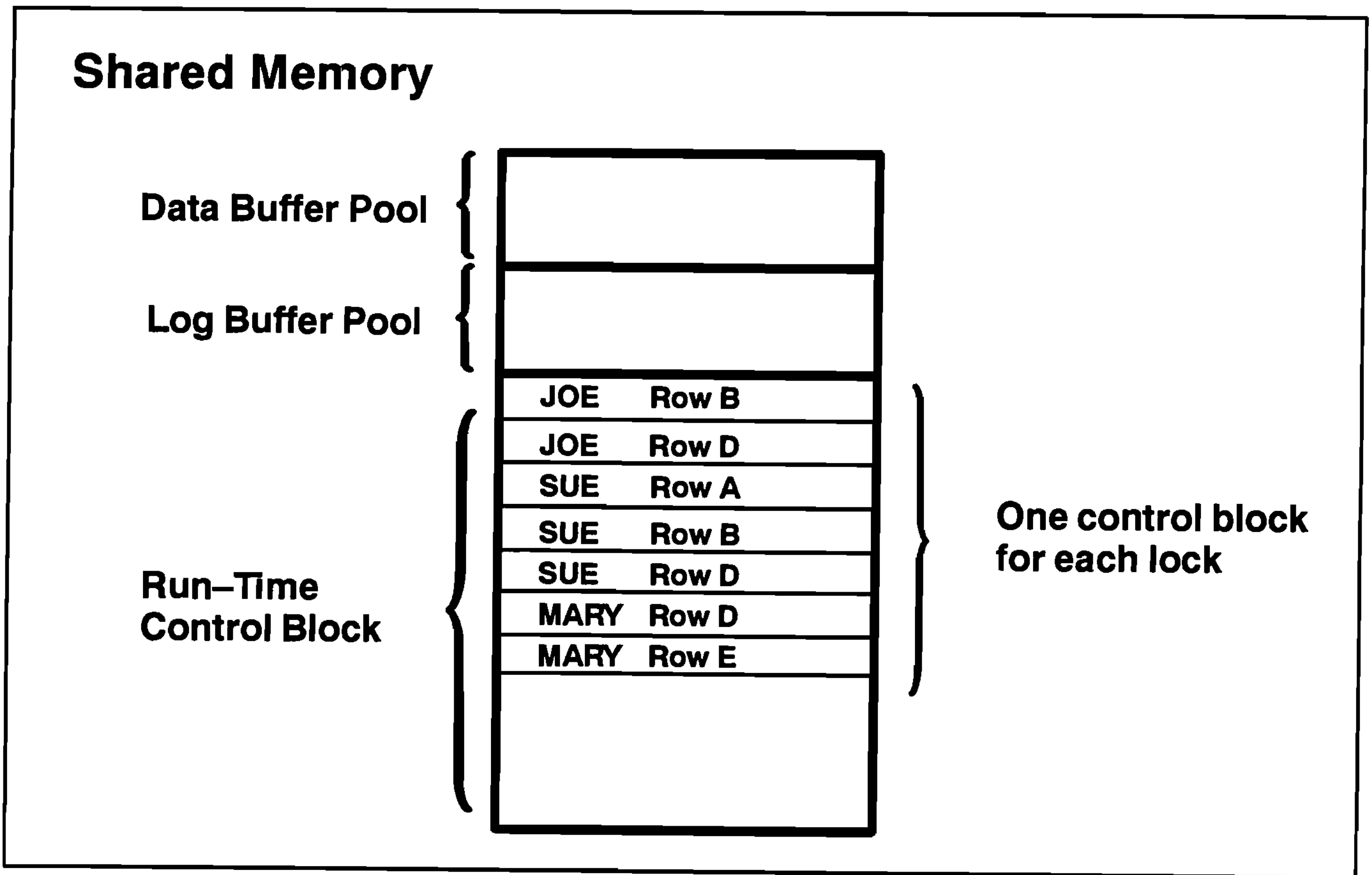


Figure 4-9.

Key Points

The following DBECon file parameters affect locking:

- Number of run-time control block pages
  - The run-time control block is an area of shared memory containing global, run-time information for the DBE. Internally, ALLBASE/SQL uses different types of control blocks to accomplish its processing. These control blocks are all allocated from the run-time control block space that has been configured for the DBE.
  - The run-time control block is allocated in memory when the database is started for the first time, using either the `START DBE` or the first `CONNECT` statement.
  - The size of the run-time control block can be specified in the `START DBE` statement, or by using the SQLUTIL `ALTDBE` command. The more concurrent activity (such as locks, transactions, page allocations, etc.) that exists in the DBE, the greater the number of run-time control block pages that are needed.
  - The default number of pages is 37. Each page is 4096 (4K) bytes.
  - Control blocks are allocated for different types of internal processing, but the majority of control blocks are used for lock management. One control block is needed for each table, page, or row lock. The greater the number of concurrent locks held, the greater the number of run-time control block pages that are needed to manage these locks. A program that manages locks well is less likely to deplete the amount of shared memory available.
  - If the run-time control block is too small, ALLBASE/SQL is not able to allocate necessary control blocks when a transaction executes an SQL statement. The transaction is rolled back, and an error is returned. In order to increase the amount of run-time control block space, the DBEnvironment must be stopped and re-started using a larger value.
- User mode
  - Default is SINGLE user mode, which means that only the user who starts the DBEnvironment can access it. MULTI user mode allows multiple users to access the DBEnvironment.
  - In SINGLE user mode, ALLBASE/SQL allocates the run-time control block from the heap of the process that started the database, rather than shared memory as was described above. Lock control blocks are acquired out of this heap by the transaction when SQL statements are executed, but only table level locks are aquired.
  - Fewer run-time control block pages are needed in SINGLE user mode than in MULTI user mode.
  - SINGLE user mode is good for testing a new DBEnvironment, and also for performing maintenance functions.
  - The user mode can be specified in the `START DBE` statement, or by using the SQLUTIL `ALTDBE` command.

Purpose of Slide

To identify DBECon file parameters associated with locking (continued).

DDL Statements in ALLBASE/SQL

ADD DBEFILE
ADD TO GROUP
ALTER DBEFILE
ALTER TABLE
CREATE DBEFILE
CREATE DBEFILESET
CREATE GROUP
CREATE INDEX
CREATE PROCEDURE
CREATE RULE
CREATE SCHEMA
CREATE TABLE
CREATE TEMPSPACE
CREATE VIEW
DROP DBEFILE
DROP DBEFILESET
DROP GROUP
DROP INDEX
DROP MODULE
DROP PROCEDURE
DROP RULE
DROP TABLE
DROP TEMPSPACE
DROP VIEW
GRANT
REMOVE DBEFILE
REMOVE FROM GROUP
REVOKE
TRANSFER OWNERSHIP
UPDATE STATISTICS

Figure 4-10.

Key Points

- DDL Enabled
  - Default is YES, which allows data definition language (DDL) statements (such as CREATE, DROP, and so on) to be issued within the DBEnvironment. DDL statements obtain exclusive locks on data in the system catalog. These locks can cause severe concurrency problems with other transactions. When the DDL enabled parameter is set to NO, DDL is disabled, which means that these statements cannot be issued successfully (an error is returned if an attempt is made to issue them). Therefore, only share locks will be obtained on the majority of tables in the system catalog.
  - When DDL has been disabled, invalid sections can still be revalidated. When revalidation occurs, exclusive locks are obtained on several tables in the system catalog; these locks can cause concurrency problems with other transactions.
    - If invalid sections exist in your DBEnvironment, you can either issue the VALIDATE statement to manually perform revalidation, or you can rely on ALLBASE/SQL to automatically revalidate the sections when they are encountered.
    - It is usually better to use the VALIDATE statement rather then to rely on automatic revalidation, because concurrency problems during production hours can be avoided. For best performance, no invalid sections should exist during high-access periods for the DBEnvironment.
    - Use the VALIDATE statement immediately after sections have become invalid (that is, when something that a section depends on is modified). For example, if an UPDATE STATISTICS statement is issued or if an index is dropped and recreated, a VALIDATE statement should also be issued because some sections might have become invalid. For best performance, issue all the DDL statements (such as UPDATE STATISTICS), then issue all the VALIDATE statements. This ensures that you only invalidate and revalidate a section once, even if it is dependent on several tables.
  - When DDL has been disabled, ALLBASE/SQL retains sections in user memory between transactions. This means that an application program that re-executes the same sections again and again does not require ALLBASE/SQL to read the sections in from disk each time. This can have a significant positive effect on performance.
  - When DDL has been disabled, certain system catalog information is retained in shared memory, which also improves performance.

Purpose of Slide

To describe lock granularity.

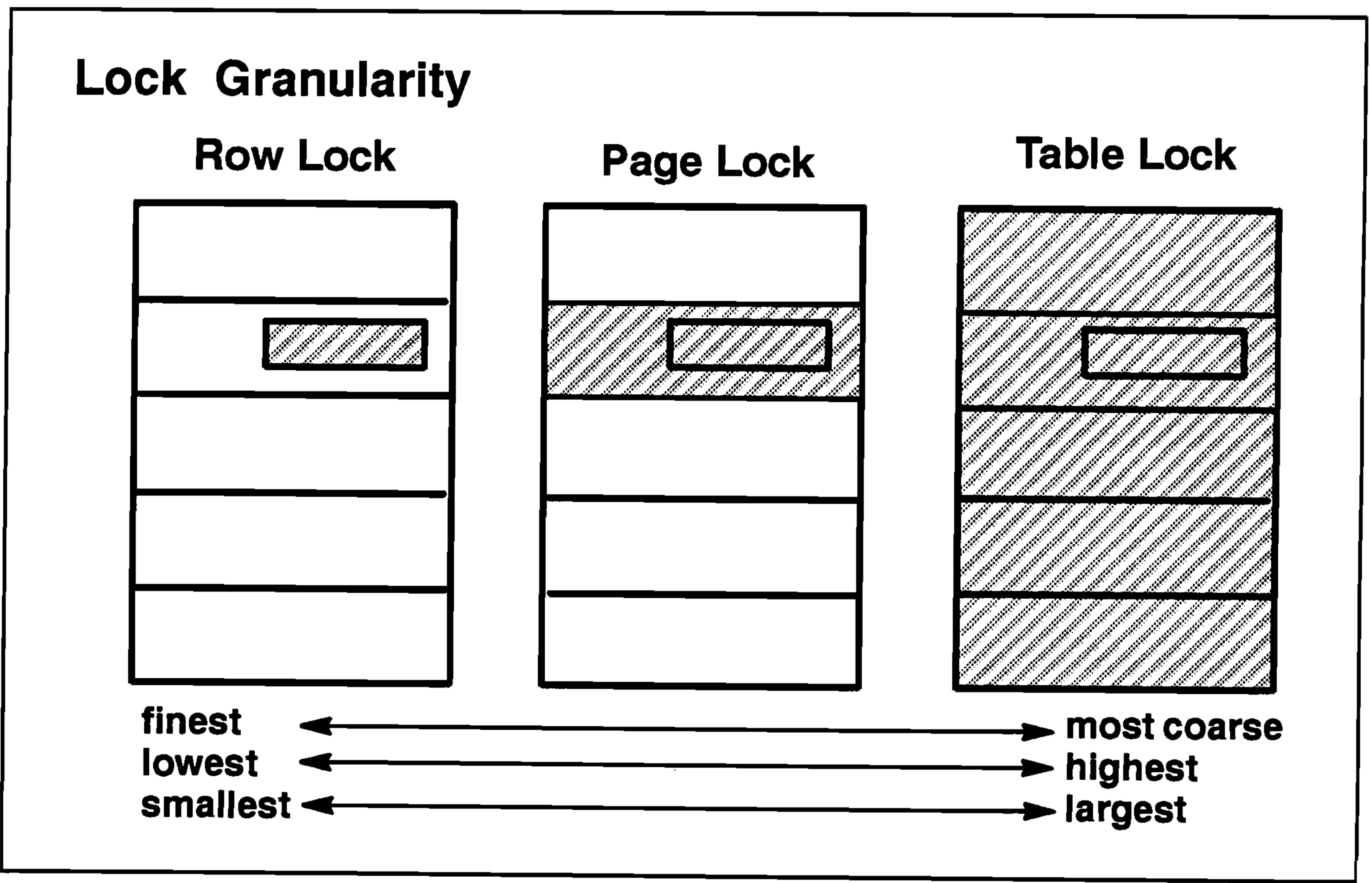


Figure 4-11.

Key Points

- **Granularity** is the size of the object that is locked. ALLBASE/SQL locks data at three levels of granularity: row (or tuple), page, and table.
- The smallest (lowest, finest) granularity is at the row level. The largest (highest, most coarse) granularity is at the table level.
- Generally, the smaller the lock granularity, the greater the number of users that can simultaneously access data in the table, because a smaller portion of the data is locked by each user.
- When any SQL statement is executed, page level locks are acquired on one or more system catalog tables (that is, tables owned by the special user HPRDBSS). In addition, row level locks are acquired on certain ALLBASE/SQL internal tables (that is, tables owned by DBCore). You cannot directly change the locking behavior of these tables.
- When SQL statements that reference a user table are executed, row, page, or table locks of different kinds may be obtained on the table. You can help control the granularity of locking by doing the following:
  - You can modify the implicit locking structure of a table by changing the table's type with the **ALTER TABLE** statement. Four table types exist (these will be explained in greater detail later in this module). The following locks are acquired by default when data is accessed in each type of table:
    - PRIVATE (default) - table locks
    - PUBLICREAD - table locks
    - PUBLIC - page locks
    - PUBLICROW - row locks
  - You can use the **LOCK TABLE** statement to override the implicit locking structure of a table for a given transaction. For example, if a transaction will read every row in a PUBLICROW table, you can use the **LOCK TABLE** statement to obtain a single table lock instead of many row locks. **LOCK TABLE** will also be explained in greater detail later in this module.

Purpose of Slide

To describe lock granularity (continued).

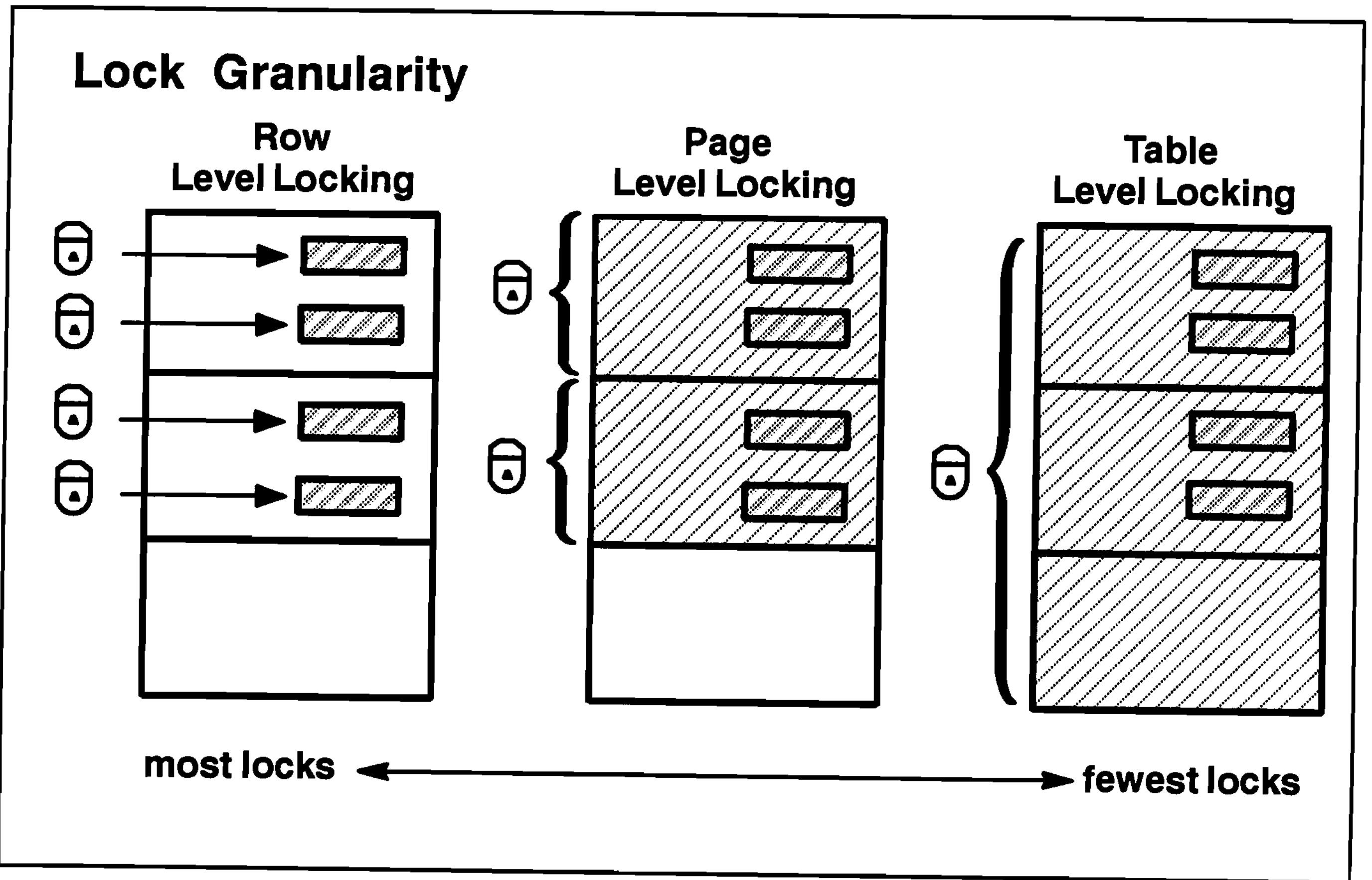


Figure 4-12.

Key Points

- Page level locking uses more run-time control block space than table level locking, since each page must be locked. Row level locking uses even more space than page level locking, since each row must be locked. This can require a considerable number of lock control blocks (and pages of shared memory). The following chart indicates the maximum number of locks that could be obtained on a table by using table, page, and row level locking (the maximum number is possible when all of the rows in the table are locked):

```
Locking Level | Maximum number of locks
--------------|------------------------
Table level   | 1
Page  level   | n + (1)
Row   level   | m + (n + 1)

n equals the number of pages in the table.
m equals the number of rows in the table.
```

- Table level locking requires 1 lock. Page level locking requires up to n page level locks, plus 1 intention lock at the table level (intention locks will be explained later in this module). Row level locking requires up to m row level locks, and up to (n + 1) intention locks at the page and table level.
- Because row level locking on a large table may consume a tremendous number of run-time control block pages, the use of the PUBLICROW table type on large tables is discouraged. Large tables for which maximum concurrency is desired should generally be defined as PUBLIC. The PUBLICROW table type should generally be reserved for use on small tables.

Purpose of Slide

To describe the SHARE and EXCLUSIVE lock mode types.

Fundamental Locking Requirements

SELECT
FETCH } => READ ⟶ SHARE (S) LOCK

INSERT
UPDATE
DELETE } => WRITE ⟶ EXCLUSIVE (X) LOCK

Figure 4-13.

Key Points

- There are two basic requirements of locking:
  - READ operations (such as SELECT and FETCH), must acquire SHARE locks before rows can be retrieved.
  - WRITE operations (such as UPDATE, INSERT, and DELETE), must acquire EXCLUSIVE locks before rows can be modified.
- A SHARE (S) lock permits reading by other users. No other transaction may modify the data that is locked with an S lock.
  - When an S lock is obtained at the table level, the transaction can read all rows in the table. No row or page level locks are acquired when the transaction reads a row (the S lock at the table level covers all of the rows in the table, so additional locks are not neccessary).
  - When an S lock is obtained at the page level, the transaction can read all rows on the page. No row level locks are acquired when the transaction reads a row (the S lock at the page level covers all of the rows on the page).
  - When an S lock is obtained at the row level, the transaction can read the row.
- An EXCLUSIVE (X) lock prevents access by any other user. An X lock is the strongest type of lock. No other transaction may read or modify the data that is locked with an X lock. An X lock must be obtained (either at the table, page, or row level) when user data is updated, inserted, or deleted.
  - When an X lock is obtained at the table level, the transaction can read and modify all rows in the table. No row or page level locks are acquired when the transaction reads or modifies a row.
  - When an X lock is obtained at the page level, the transaction can read and modify all rows on the page. No row level locks are acquired when the transaction reads or modifies a row.
  - When an X lock is obtained at the row level, the transaction can read and modify the row.

Purpose of Slide

To describe intention locking.

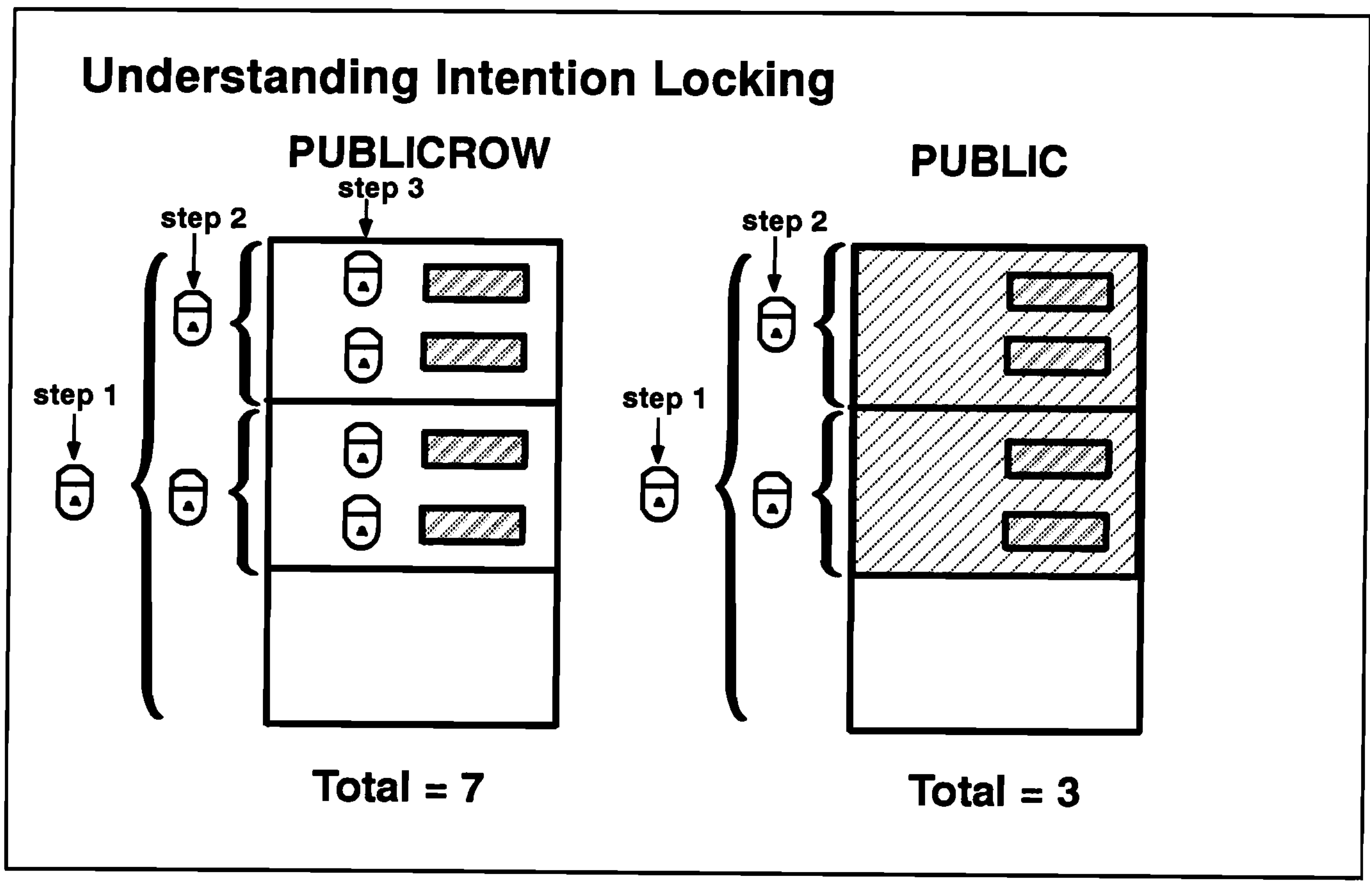


Figure 4-14.

Key Points

- Intention locks reduce the number of locks that must be examined when a new lock is allocated. Intention locks allow transactions to quickly determine if rows (or pages) in a given table have been locked by other transactions. Without the use of intention locks, ALLBASE/SQL would have to search all row (or page) locks on a table to determine whether or not a new lock request could be granted. Such searching would be very inefficient, especially on large tables.
- When a lock is acquired at a smaller granularity, an intention lock is first acquired at a larger granularity.
    - When a row lock is acquired on a PUBLICROW table, the following occurs:
        - An intention lock is acquired on the table.
        - After the table lock has been granted, an intention lock is acquired on the page.
        - After the page lock has been granted, the row lock is acquired.
    - When a page lock is acquired on a PUBLIC table, the following occurs:
        - An intention lock is acquired on the table.
        - After the table lock has been granted, the page lock is acquired.
- When two locks are compatible, both access requests are allowed to the data object (that is, the table, page or row) at the same time. When compatible locks exist on a data object, ALLBASE/SQL computes which lock is strongest and stores this information. When locks are not compatible, the second access request must wait until the lock acquired by the first access request is released.
- Any request for a lock at the table level is compared to the strongest lock on the table. If the table has already been locked in an incompatible mode by another transaction, the transaction that is requesting the lock will wait until the lock can be granted (or a timeout occurs). Next, a request for a page lock is made. If this page has been locked in an incompatible mode by another transaction, the requesting transaction will wait until the lock can be granted (or a timeout occurs). Finally, if row level locking is used, the request for a row lock is made.
- A transaction timeout will only occur if the wait for a single database resource consumes an amount of time equal to the timeout limit. If a transaction's timeout limit is set to 10 seconds, and it takes a 5 second wait to obtain an intention lock on the table and a 6 second wait to obtain a lock on a page, the transaction will not timeout. If either wait consumed 10 seconds, then a timeout would occur.
- Intention locks are only acquired on PUBLICROW and PUBLIC tables. Remember that PUBLICROW tables use row level locking, and PUBLIC tables use page level locking.
- PRIVATE and PUBLICREAD tables use table level locking, so intention locks are not needed by ALLBASE/SQL.
- The three types of intention locks are INTENT SHARE (IS), INTENT EXCLUSIVE (IX), and SHARE and INTENT EXCLUSIVE (SIX).

Purpose of Slide

To describe the INTENT SHARE (IS) lock mode type.

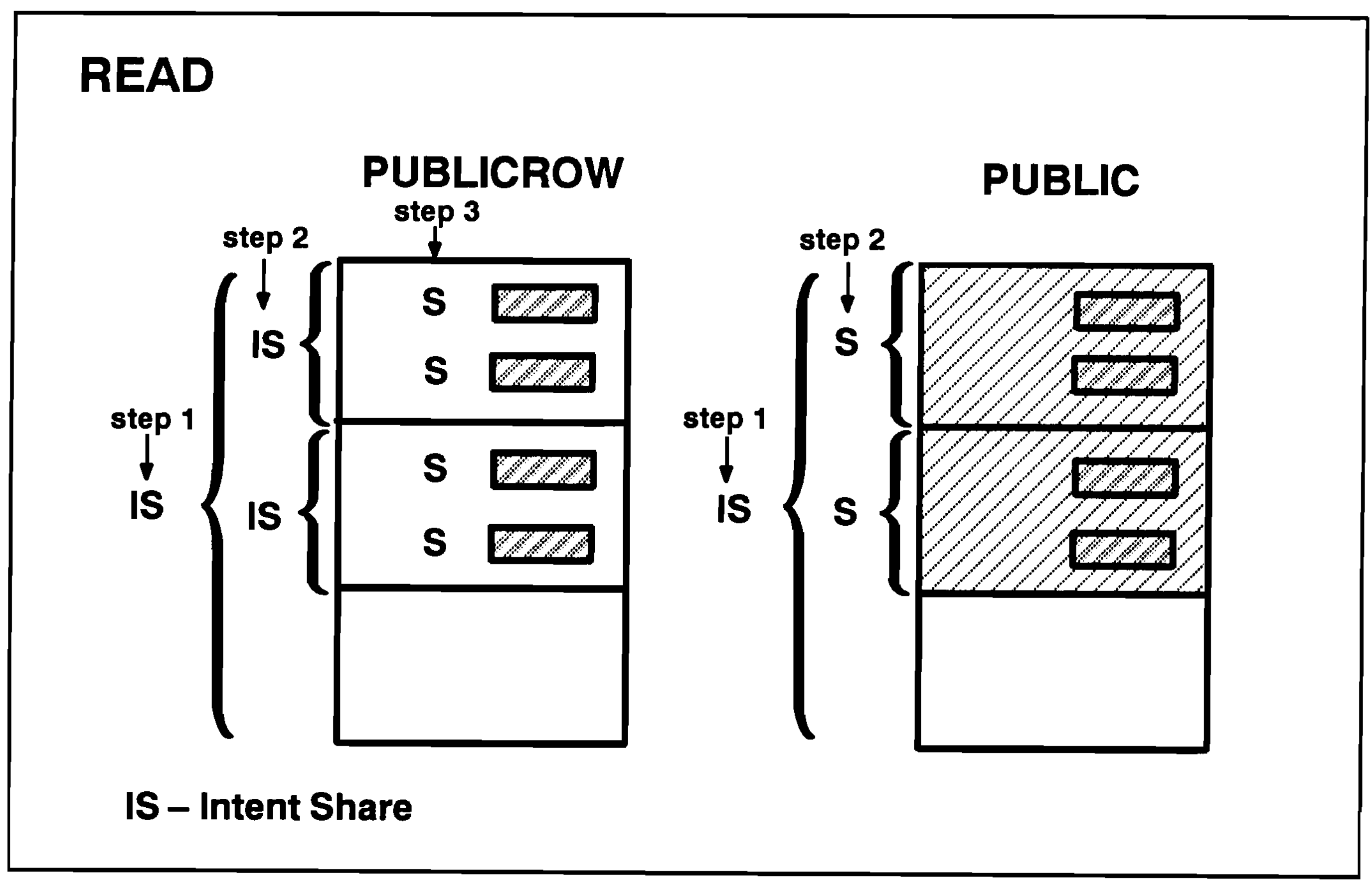


Figure 4-15.

Key Points

- An intent share (IS) lock indicates an intention to read data (that is, acquire an S lock) at a lower level of granularity.
  - An IS lock on a PUBLICROW table together with an IS lock on a page indicates an intention to read one or more rows on that page.
  - An IS lock on a PUBLIC table indicates an intention to read one or more pages.
  - An IS lock on a table indicates that the transaction wants to read *some* of the rows in the table (an S lock on a table indicates that the transaction wants to read *all* of the rows).
  - An IS lock is also called a subshare lock.

Purpose of Slide

To describe the INTENT EXCLUSIVE (IX) Lock Mode Type.

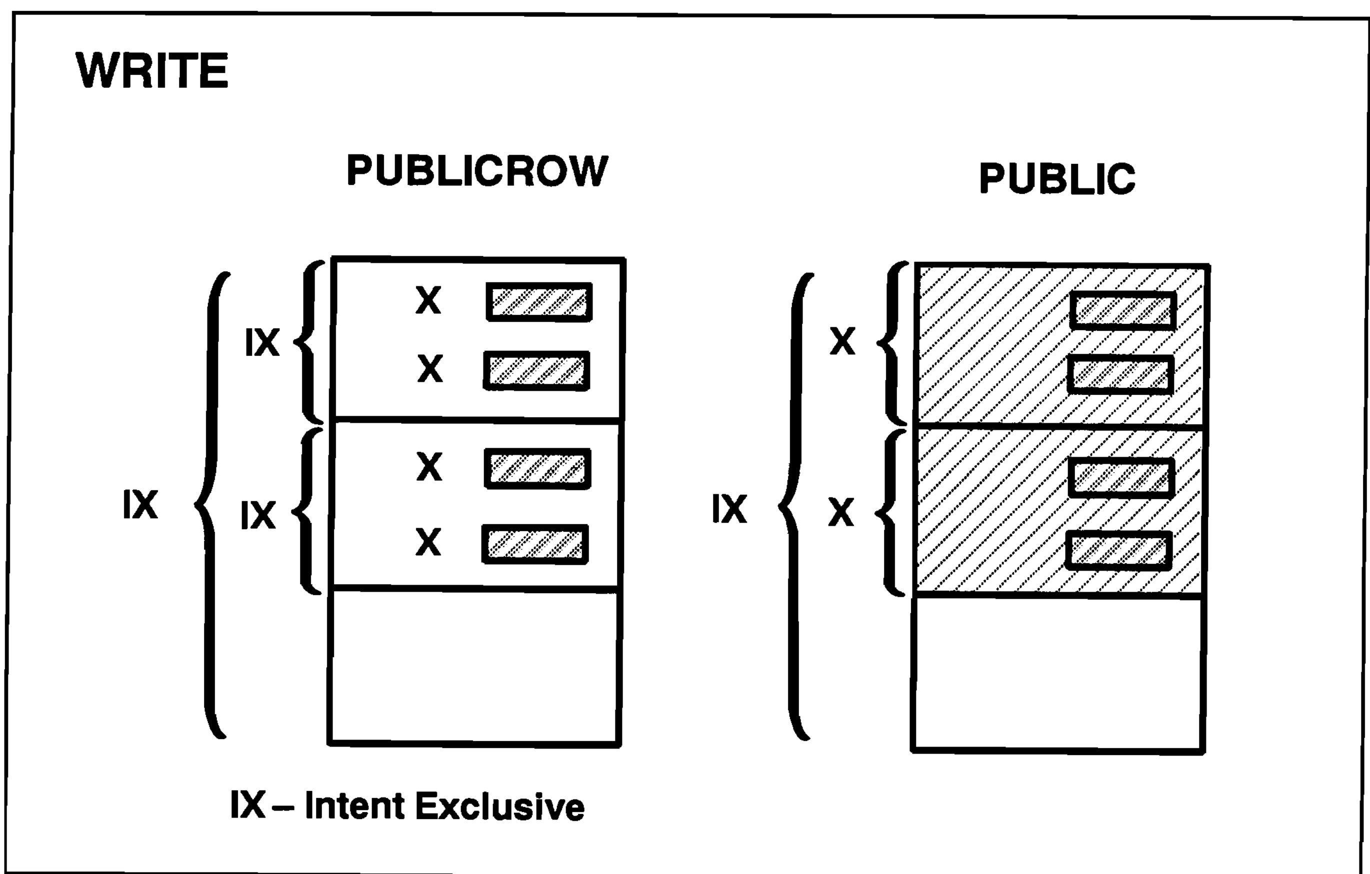


Figure 4-16.

Key Points

- An intent exclusive (IX) lock indicates an intention to write data (that is, acquire an X lock) at a lower level of granularity.
  - An IX lock on a PUBLICROW table together with an IX lock on a page indicates an intention to write to one or more rows on that page.
  - An IX lock on a PUBLIC table indicates an intention to write to one or more pages.
  - An IX lock on a table indicates that the transaction wants to write to *some* of the rows in the table (an X lock on a table indicates that the transaction wants to write to *all* of the rows).
  - An IX lock is also called a subexclusive lock.

Purpose of Slide

To describe the SHARE and INTENT EXCLUSIVE (SIX) lock mode type.

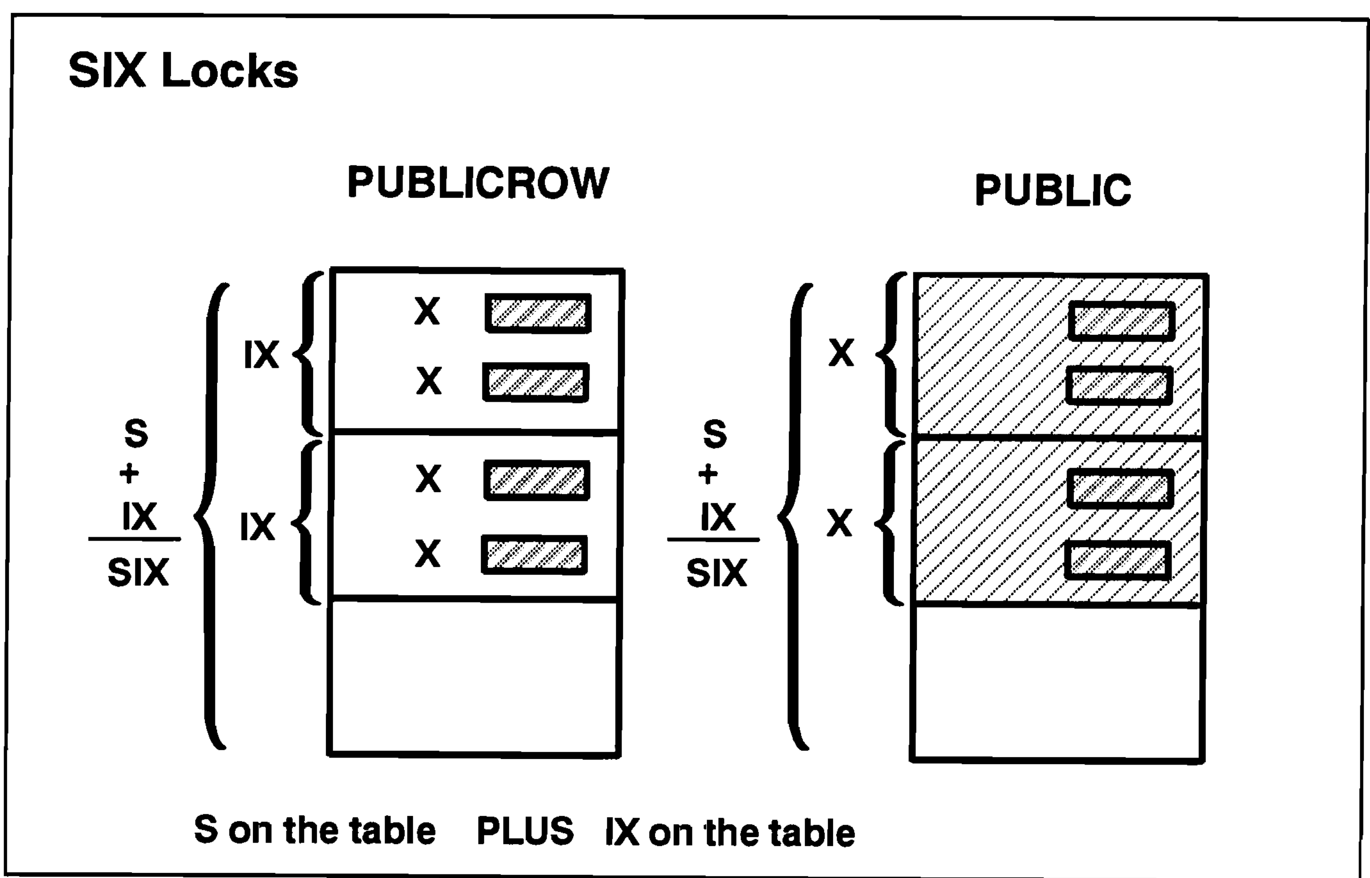


Figure 4-17.

Key Points

- A share and intent exclusive lock (or SIX lock, pronounced as the separate letters S I X rather than like the number six) indicates an S lock at the current level plus an intention to write data at a lower level of granularity. Think of an SIX lock as an S lock plus an IX lock. Only one transaction can be granted an SIX lock on a table at a time.
  - An SIX lock on a PUBLICROW table indicates an intention to read all of the rows in the table and to write to a few. Rows that are read but not updated will not obtain S locks (the S lock in the SIX lock at the table level covers all of the rows). Rows that are updated will obtain X locks, but only after IX intention locks have been obtained on the pages that contain them.
  - An SIX lock on a PUBLIC table indicates an intention to read all of the pages in the table and to write to a few. Pages that are read but not updated will not obtain S locks. Pages that are updated will obtain X locks.
  - Occasionally, an SIX lock is acquired on a row of a PUBLICROW table, or on a page of a PUBLIC table. This occurs when the transaction has read the row or page with the intention of writing to it later. An SIX lock on a row or page must be converted to an X lock before the actual update may occur. No other transaction can read or modify a row or page that has been locked with an SIX lock.
  - An SIX lock is stronger than an S lock or an IX lock. When a transaction obtains an SIX lock on a table, only that transaction will be able to modify data in the table. In this respect, an SIX lock slightly resembles an X lock. With an SIX lock, however, other transactions that want to read *some* of the data (read data at the row or page level and obtain an IS lock on the table) are allowed to proceed, so concurrency is better than with an X lock. Lock mode compatibility will be described in greater detail later in this module.
  - If other transactions obtain S row locks in a PUBLICROW table or S page locks in a PUBLIC table on rows that the SIX transaction wants to modify, the SIX transaction must wait until the S locks are released before it can modify the data.
  - Other transactions that want to read all of the data (obtain an S lock on the table) or that want to write to any portion of the data are not allowed to proceed until the SIX lock is released.
  - An SIX lock is also called a share subexclusive lock.

Purpose of Slide

To review intention locking on a PUBLICROW table.

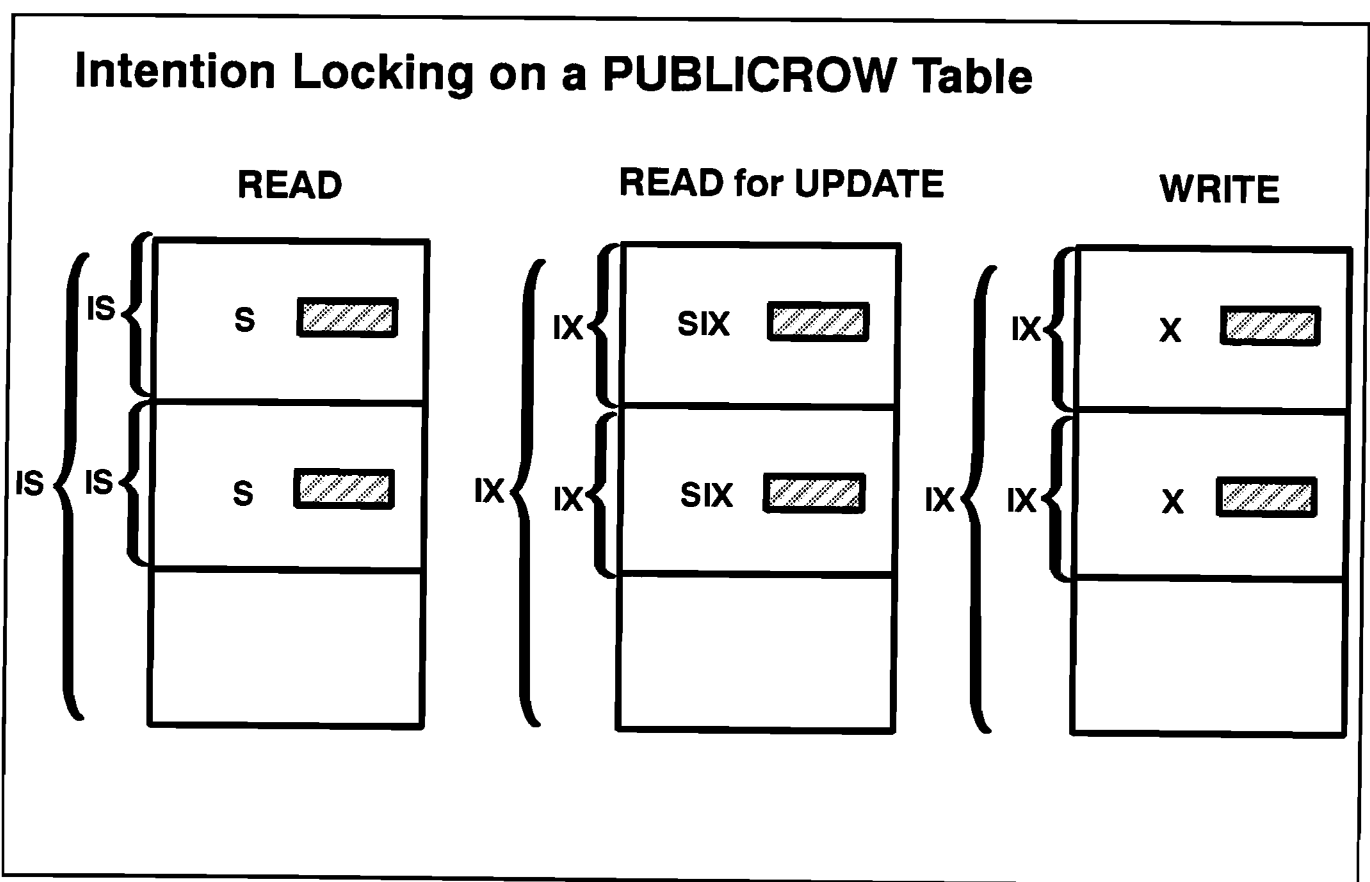


Figure 4-18.

Key Points

- The following locks are obtained by default when a read or a write is made to a single row of a PUBLICROW table:
  - READ:
    - An IS lock is obtained on the table.
    - An IS lock is obtained on the page.
    - An S lock is obtained on the row.
  - READ with an intention to write (such as a `REFETCH` statement, or a `DECLARE CURSOR FOR UPDATE` followed by an `OPEN` and a `FETCH`):
    - An IX lock is obtained on the table.
    - An IX lock is obtained on the page.
    - An SIX lock is obtained on the row (remember that no other transaction can read or modify a row or page that has been locked with an SIX lock).
  - WRITE:
    - An IX lock is obtained on the table.
    - An IX lock is obtained on the page.
    - An X lock is obtained on the row.

Purpose of Slide

To review intention locking on a PUBLIC table.

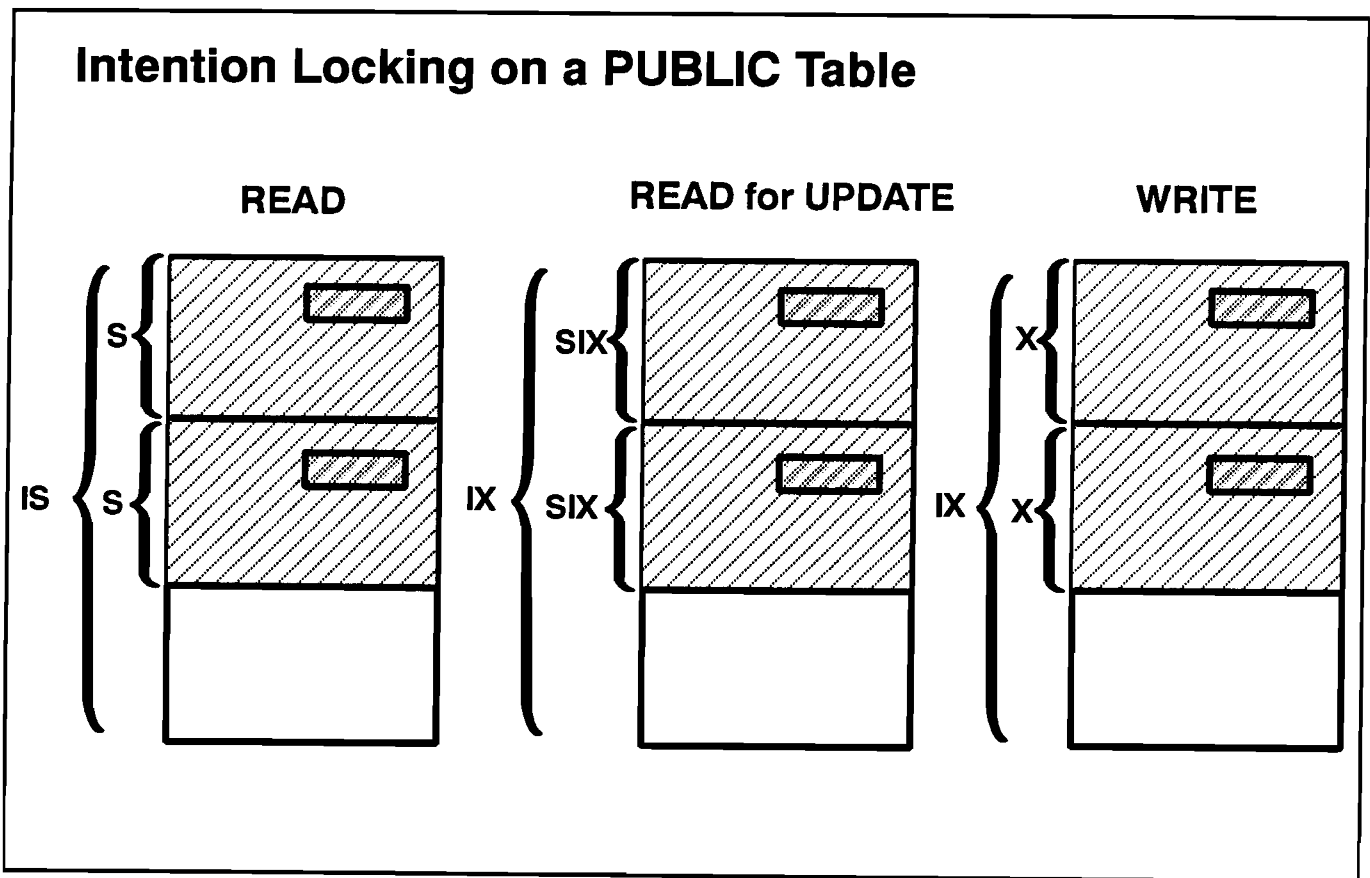


Figure 4-19.

Key Points

- The following locks are obtained by default when a read or a write is made to a single row of a PUBLIC table:
  - READ:
    - An IS lock is obtained on the table.
    - An S lock is obtained on the page.
  - READ with an intention to write:
    - An IX lock is obtained on the table.
    - An SIX lock is obtained on the page.
  - WRITE:
    - An IX lock is obtained on the table.
    - An X lock is obtained on the page.

Purpose of Slide

To describe how to choose table types to control locking.

Locking Behavior of ALLBASE/SQL Tables

Table Type	Locks Obtained for READ	Locks Obtained for WRITE
PRIVATE (default)	Table X	Table X
PUBLICREAD	Table S	Table X
PUBLIC	Table IS Page S	Table IX Page X
PUBLICROW	Table IS Page IS Row S	Table IX Page IX Row X

Figure 4-20.

Key Points

- The first parameter in the CREATE TABLE statement specifies the implicit locking structure. You may use the ALTER TABLE statement to permanently change the structure associated with a table.
- The four implicit locking structures and their normal locking behaviors are:
    - PRIVATE (default)
        - Locks at the table level (X) for reads.
        - Locks at the table level (X) for writes.
        - Allows one transaction at a time to read from or write to the table. PRIVATE tables reduce overhead and shared memory needs for ALLBASE/SQL (since row or page locks are never acquired), and are less likely to cause a deadlock condition because each table is always accessed exclusively by one user. However, they severely reduce concurrency.
    - PUBLICREAD
        - Locks at the table level (S) for reads.
        - Locks at the table level (X) for writes.
        - Allows only one transaction at a time to write to the table. When a transaction is writing to the table, no other transaction can access it. When no transaction is writing to the table, multiple transactions may read from it at the same time. PUBLICREAD tables also reduce overhead and shared memory needs for ALLBASE/SQL (since row or page locks are never acquired), but they provide better concurrency than PRIVATE tables. Tables that are rarely updated should generally be PUBLICREAD.
    - PUBLIC
        - Locks at the page level (S) for reads. Intention locks are generated at the table level.
        - Locks at the page level (X) for writes. Intention locks are generated at the table level.
        - Allows multiple read transactions and write transactions to execute at the same time on the table. Page level locking is used, which uses more shared memory than PRIVATE or PUBLICREAD, but less shared memory than PUBLICROW. Large tables for which maximum read and write concurrency is desired should generally be PUBLIC.
    - PUBLICROW
        - Locks at the row level (S) for reads. Intention locks are generated at the table level and at the page level.
        - Locks at the row level (X) for writes. Intention locks are generated at the table level and at the page level.
        - Allows multiple read transactions and write transactions to execute at the same time on the table. Row level locking is used. Small PUBLICROW tables are less likely to have deadlocks then small PUBLIC tables. Small tables for which maximum read and write concurrency is desired should generally be PUBLICROW.

Purpose of Slide

To demonstrate the LOCK TABLE statement.

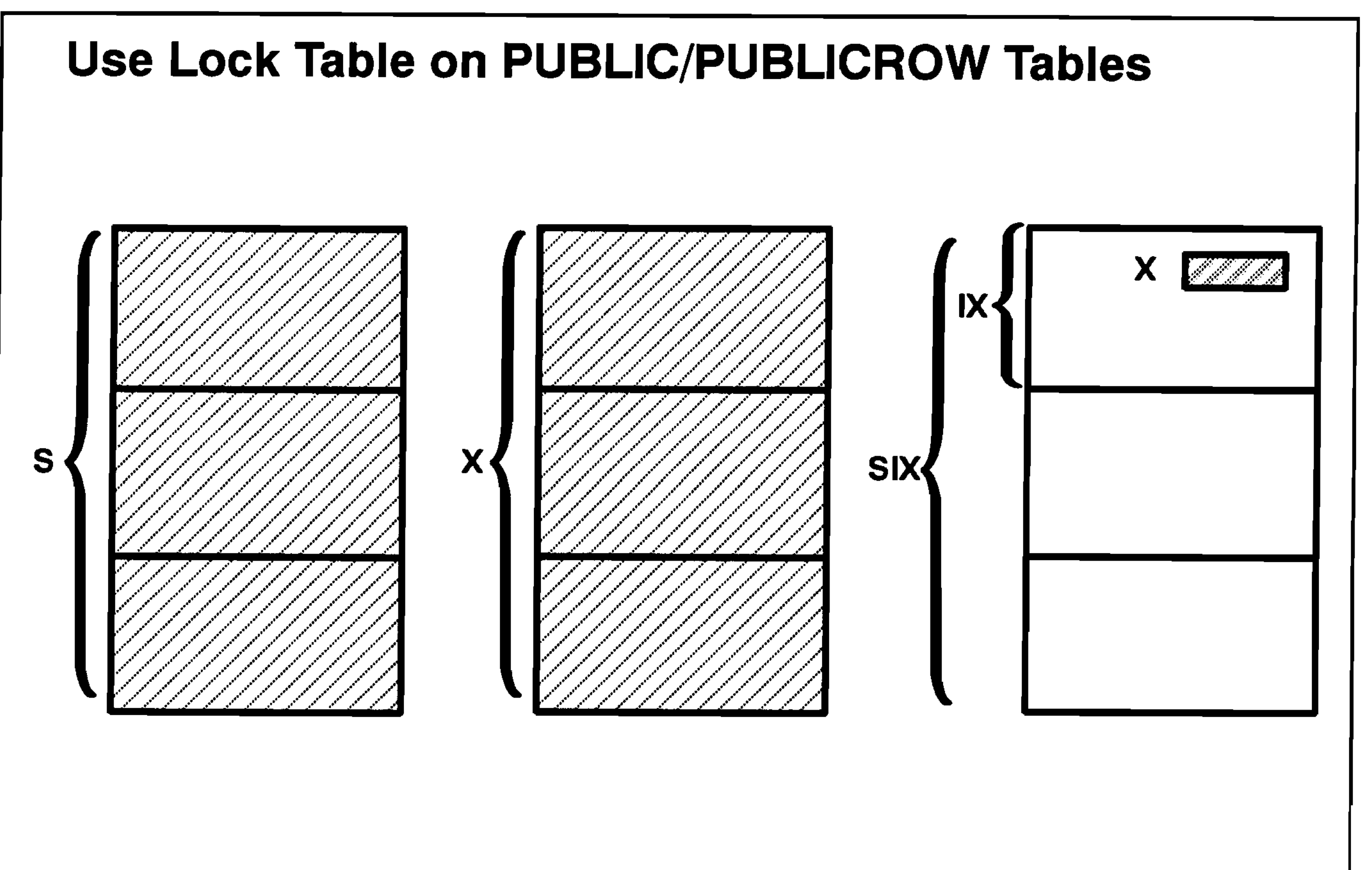


Figure 4-21.

Key Points

- When data is accessed in a table, ALLBASE/SQL generates appropriate locks by using the implicit locking structure that has been established for the table. The implicit locking structure is originally established using the `CREATE TABLE` statement, but it can be changed by using the `ALTER TABLE` statement.
- You can use the `LOCK TABLE` statement to override implicit locking in some situations.
- To use the `LOCK TABLE` statement, you must have OWNER or SELECT authority for the table, or DBA authority.
- All locks acquired by the `LOCK TABLE` statement are released when the transaction terminates.
- The `LOCK TABLE` statement explicitly locks tables in three modes:
  - SHARE (S)
  - SHARE UPDATE (SIX)
  - EXCLUSIVE (X)
- The following are good uses of the `LOCK TABLE` statement on a PUBLICROW or PUBLIC table:
  - If a query will read most or many rows in the table, it may be useful to obtain an S lock at the table level to minimize deadlocks and shared memory needs. When the table is locked in SHARE mode, other users can not modify it for the duration of the transaction. Also, row and page locks are not acquired, which reduces the overhead and shared memory needs of ALLBASE/SQL.
  - If a query will write to most or many rows in the table, it may be useful to obtain an X or an SIX lock at the table level to minimize deadlocks and shared memory needs.
    - When the table is locked in EXCLUSIVE mode, other users can not access it for the duration of the transaction. Row and page locks are not acquired.
    - When the table is locked in SHARE UPDATE mode, other users can not modify it for the duration of the transaction. Row and page locks are not acquired for reads, but they are acquired for the rows or pages to which the transaction writes. An SIX lock provides greater concurrency than an X lock, but more shared memory will be used by the transaction. Compared to a query in which an X lock has been obtained, an SIX query can take longer to execute because it may need to wait to modify data that has been read (and locked with S locks) by other transactions.

Purpose of Slide

To demonstrate the LOCK TABLE statement (continued).

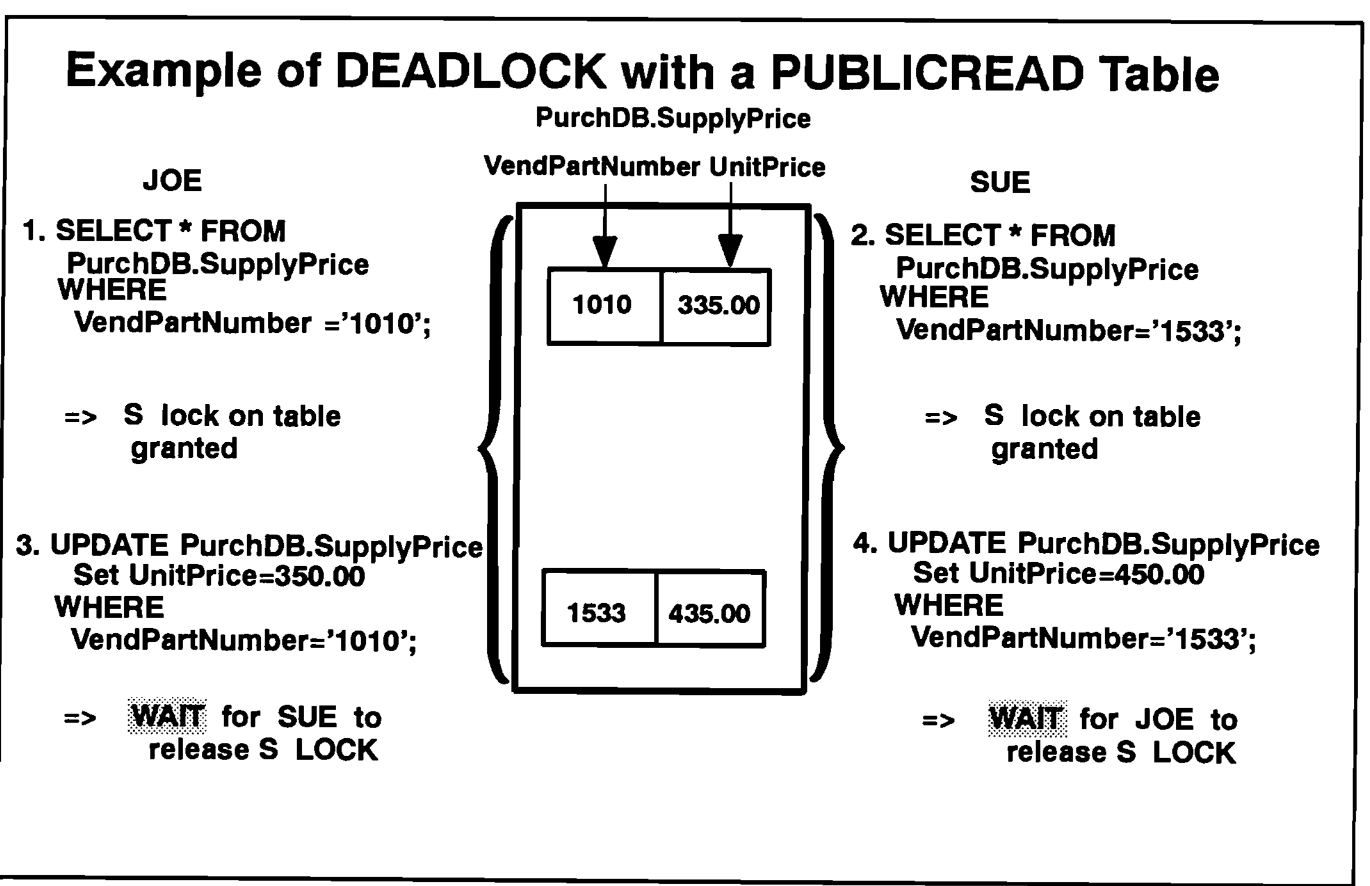


Figure 4-22.

Key Points (continued)

- The following are good uses of the `LOCK TABLE` statement on a PUBLICREAD table:
  - ☐ If a transaction initially reads from the table and later updates it, you should use the `LOCK TABLE IN EXCLUSIVE MODE` statement to obtain an X lock on the table prior to the first read. This action will help minimize deadlocks with other similar transactions. When data is read, an S lock is obtained on the PUBLICREAD table by default. Before the data can be updated, the S lock needs to be upgraded to an X lock. A deadlock situation will arise if two transactions have both obtained an S lock on a table, and both are trying to upgrade their lock to an X lock. When the table is locked with an X lock before the first read, other users can not access the table for the duration of the transaction. Other transactions that read and update the table will wait until the first transaction has completed, instead of entering into a deadlock with it.
  - The example above illustrates one of the most common mechanisms for encountering deadlocks: lock promotion. To minimize deadlocks, a transaction should avoid relying on ALLBASE/SQL to promote a weaker lock into a stronger lock.
    - ☐ A transaction should request the strongest lock that it needs before a weaker lock is obtained. In other words, if a transaction will read data and perform an update, it should acquire an X lock before it acquires an S lock (as was recommended in the example above). If a transaction will only read data, it should only acquire an S lock. You can increase the strength of the locks obtained by a transaction by using the `LOCK TABLE` command and by using the FOR UPDATE clause when a cursor is declared (this will be explained later in this module).
    - ☐ Sometimes it is useful to split an original transaction into two transactions: one that only reads data (so it only obtains S locks), and a second one that refetches data (which obtains an SIX lock) to confirm the current value of the row prior to making a modification (which obtains an X lock). A promotion from SIX to X will usually not cause a deadlock. The `REFETCH` statement will be explained later in this module.
  - You can also encounter deadlocks by obtaining locks in different orders. All transactions should use similar algorithms to obtain locks in the same order. This will minimize deadlocks.
    - ☐ For example, if one transaction obtains an exclusive lock on table A and then on table B, and another transaction obtains an exclusive lock on table B and then on table A, there is a good chance that they will enter into a deadlock.
    - ☐ If both transaction obtain locks in the same order (first on table A and then on table B), the transactions will always wait on each other, instead of deadlocking.

Purpose of Slide

To demonstrate the LOCK TABLE statement (continued).

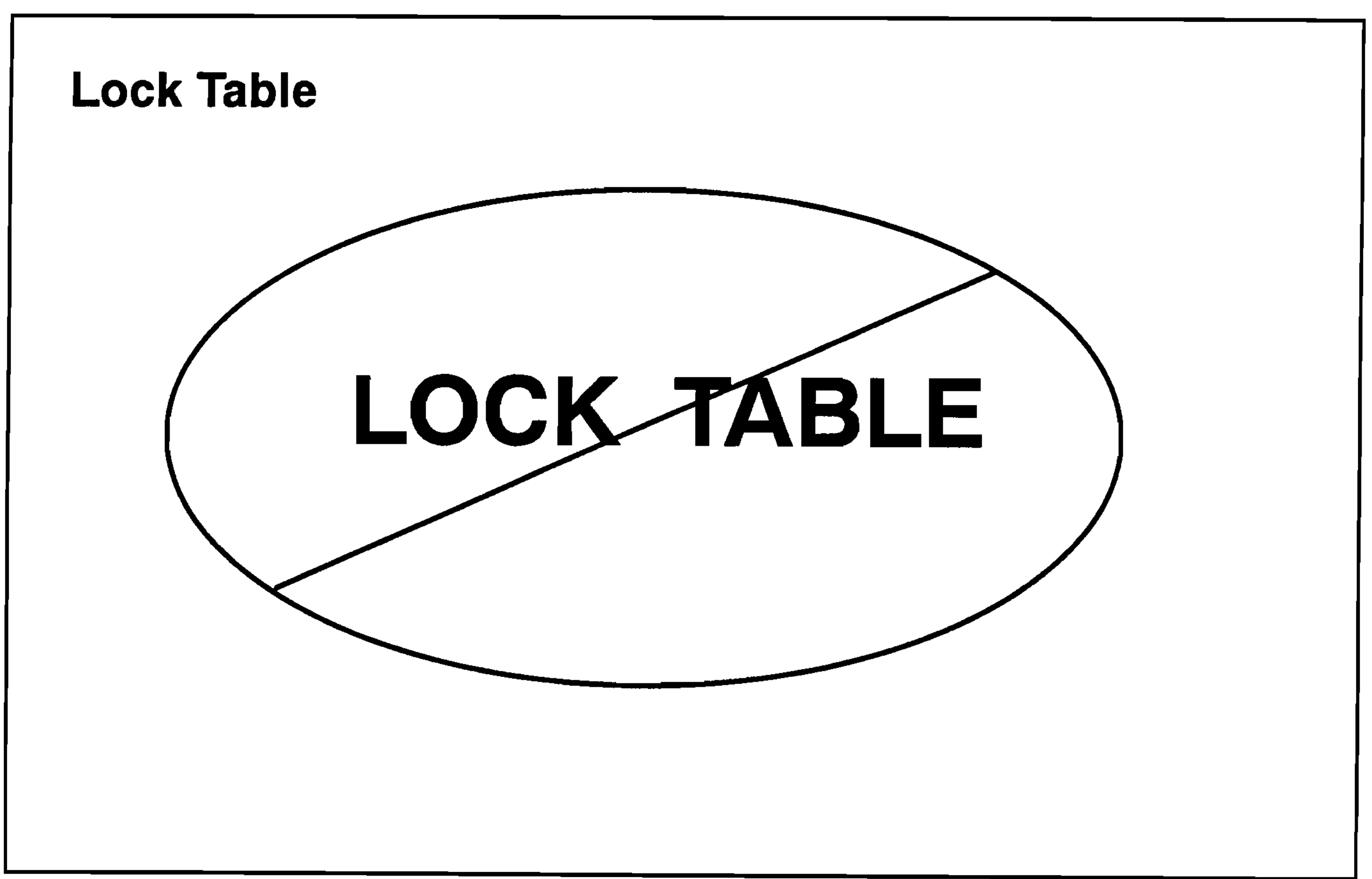


Figure 4-23.

Key Points (continued)

- You cannot weaken an implicit locking structure using the LOCK TABLE statement:
  - There is no benefit when a PRIVATE table is locked in SHARE or SHARE UPDATE mode. Your transaction will still acquire an X lock at the table level when any access to the table is made. In fact, your LOCK TABLE request will actually obtain an X lock.
  - There is no benefit when a PUBLICREAD table is locked in SHARE UPDATE mode. Your transaction will still acquire an X lock at the table level when any write to the table is made.

Purpose of Slide

To describe lock compatibility.

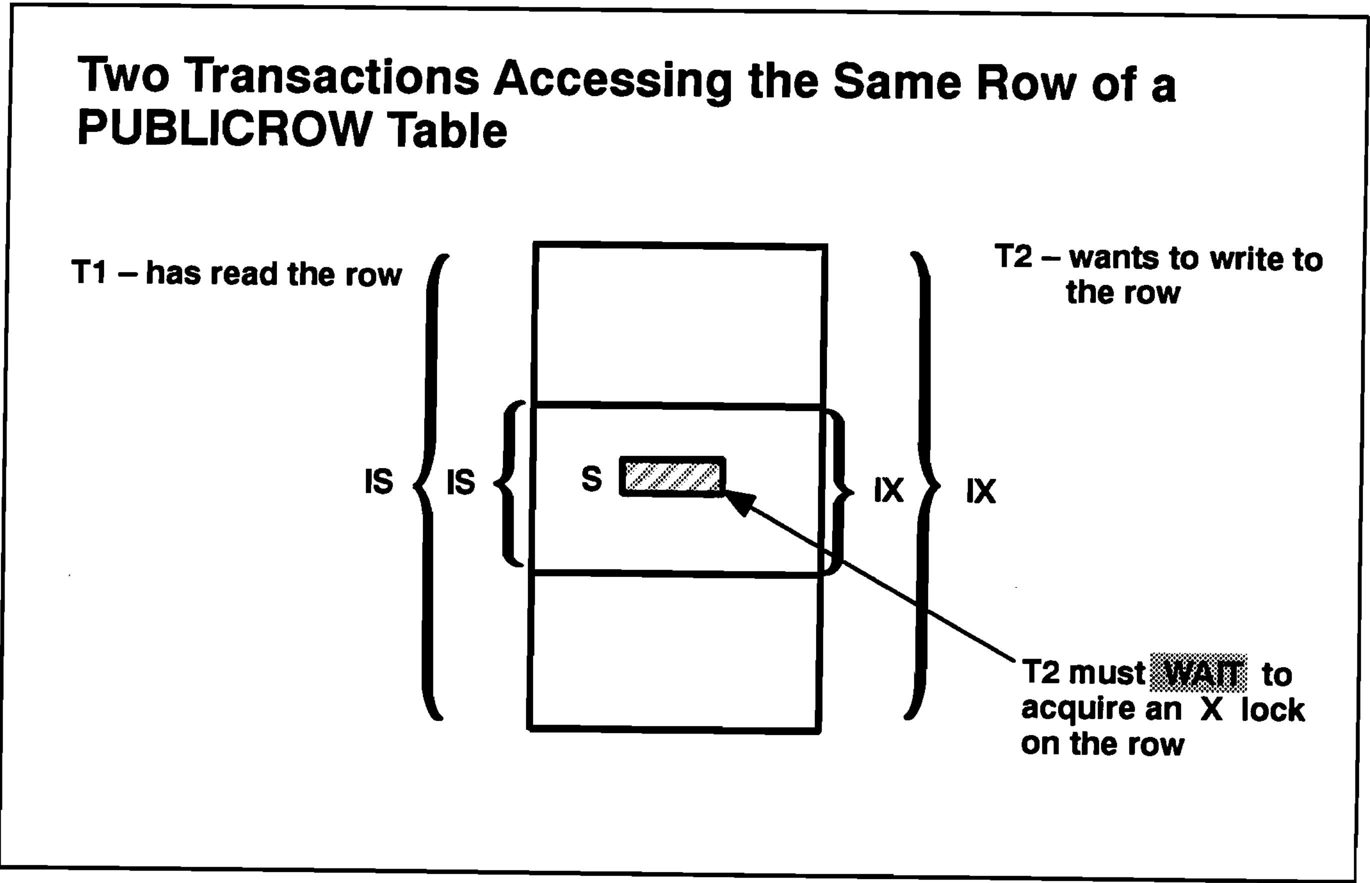


Figure 4-24.

Key Points

- The chart below shows the compatibility of different lock mode types:

LOCK THAT IS REQUESTED

LOCK THAT ALREADY EXISTS

	IS	IX	S	SIX	X
IS	*	*	*	*	
IX	*	*			
S	*		*		
SIX	*				
X					

COMPATIBLE (*) — The second lock request is granted. Both transactions are allowed to concurrently access the data object at the same time.

NOT COMPATIBLE ( ) — The second lock request must wait until the lock acquired by the first transaction is released.

- Granularity is used when determining compatibility. For example, suppose that two transactions want to access the same row in a PUBLICROW table. The first transaction is reading it, the second transaction wants to update it. The following occurs within ALLBASE/SQL:
  - First the table level intention locks are compared. The first transaction has an IS lock at the table level, the second transaction needs an IX lock at the table level. These two locks are compatible, so the IX table lock is granted.
  - Next, the page level intention locks are compared. The first transaction has an IS lock, the second transaction needs an IX lock. Again, these two locks are compatible, so the IX page lock is granted.
  - Finally, the row level locks are compared. The first transaction has an S row lock. The second transaction needs an X row lock. These two lock modes are *not* compatible, so the second transaction must wait. However, if the second transaction had wanted to write to a different row on the same page as the first row, a different X row lock could have been granted to the second transaction.

Purpose of Slide

To describe lock compatibility (continued).

Understanding Lock Compatibility

Lock that is requested

Lock that already exists	IS	IX	S	SIX	X
IS	*	*	*	*	
IX	*	*			
S	*		*		
SIX	*				
X					

COMPATIBLE (*) The second lock request is granted. Both transactions are allowed to concurrently access the data object at the same time.

NOT COMPATIBLE ( ) The second lock request must wait until the lock acquired by the first transaction is released.

Figure 4-25.

Key Points (continued)

- When referencing the chart, you may find it helpful to use the following:
  - S—indicates a transaction that wants to read *all* of the rows.
  - X—indicates a transaction that wants to write to *all* of the rows.
  - IS—indicates a transaction that wants to read *some* of the rows.
  - IX—indicates a transaction that wants to write to *some* of the rows.
  - SIX—indicates a transaction that wants to read *all* of the rows, and also write to *some* of them.
- Locks are compatible if a good chance exists that two transactions will not interfere with each other. The chart on the opposite page shows the compatibility of the lock mode types.
  - An IS lock indicates an intention to read *some* of the rows. When an IS lock has been granted, other users that want to read or write to *some* of the rows (that is, obtain an IS or IX lock) are allowed to access the data (ALLBASE/SQL assumes that the users may want to access different rows). Other users that want to read *all* of the rows (that is, obtain an S or SIX lock) are allowed. Users that want to write to *all* of the rows (that is, obtain an X lock) would interfere with the IS transaction, so they are not allowed.
  - An IX lock indicates an intention to write to *some* of the rows. When an IX lock has been granted, other users that want to read or write to *some* of the rows (IS or IX lock) are allowed. Users that want to read or write to *all* of the rows (S, SIX, or X lock) would interfere with the IX transaction, so they are not allowed.
  - An S lock indicates an intention to read *all* of the rows. When an S lock has been granted, only other users that want to read data (either *some* (IS lock) or *all* (S lock) of the rows) are allowed. Users that want to write to any portion of the data (IX, SIX, or X lock) would interfere with the S transaction, so they are not allowed.
  - An X lock indicates an intention to write to *all* of the rows. When an X lock has been granted, no other user is allowed. Users that want to access any portion of the data would interfere with the X transaction.
  - An SIX lock is equivalent to an S lock plus an IX lock. The only type of lock that is compatible with both of these locks is an IS lock. When an SIX lock has been granted, only those other users that want to read *some* of the rows (IS lock) are allowed. Users that want to read *all* of the rows (S or SIX lock) would interfere with the IX lock, and users that want to write to any portion of the data (IX, SIX, or X lock) would interfere with the S lock.

Purpose of Slide

To describe lock strengths.

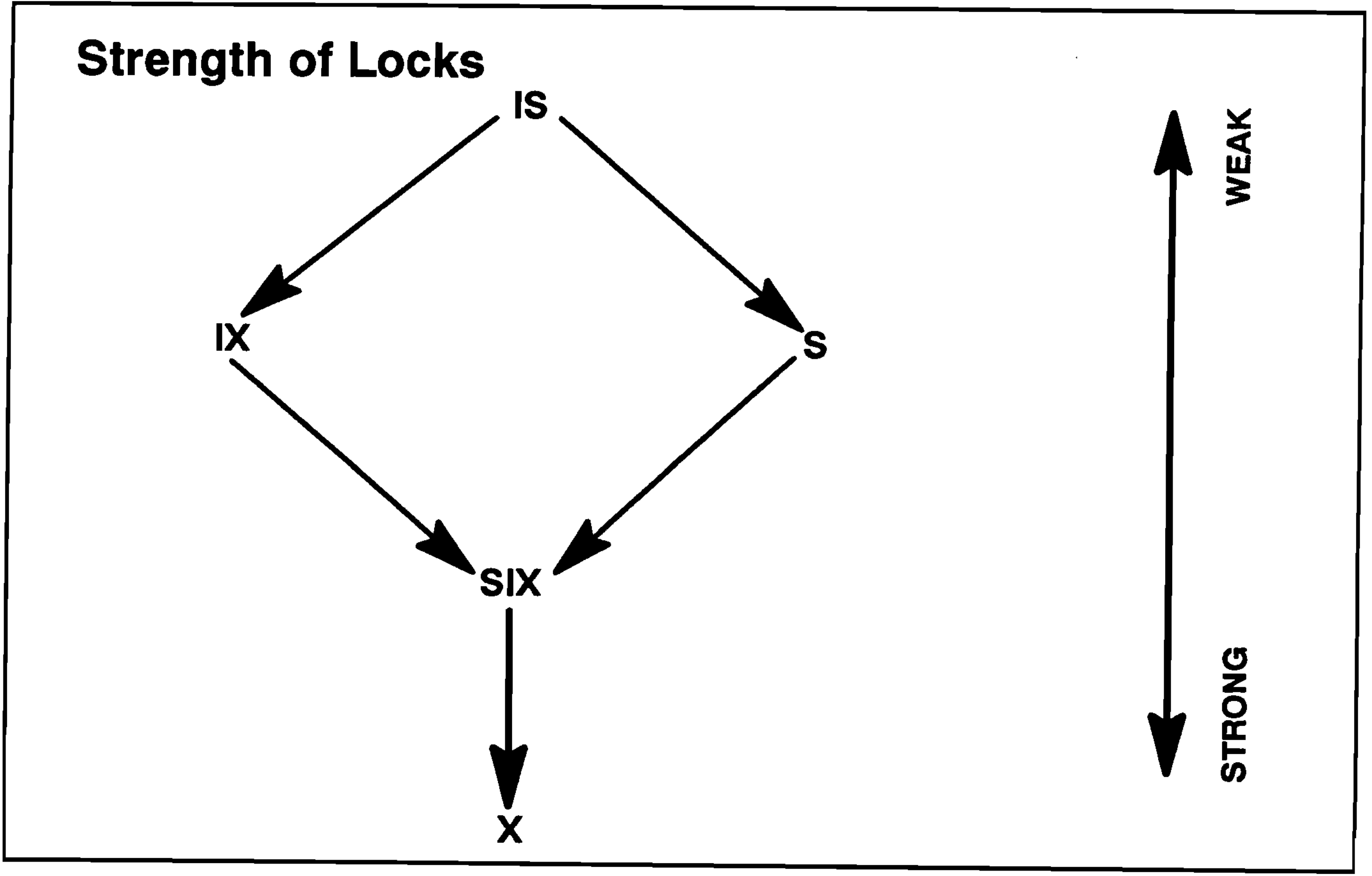


Figure 4-26.

- The **strength** of a lock refers to the number of other lock mode types with which it is compatible. Stronger locks are less compatible with other locks, and weaker locks are more compatible with other locks.
- Figure 1-26 shows the relative strength of all lock mode types. The following chart summarizes the compatiblity of each type of lock:

Lock Mode	Compatible Locks	Total
IS	IS, IX, S, SIX	4
IX	IS, IX	2
S	IS, S	2
SIX	IS	1
X		0

Key Points

- IS locks are the weakest locks, and X locks are the strongest.
- A lock can be converted from a weaker lock into a stronger lock, but never the reverse.
- S locks cannot be converted into IX locks, and IX locks cannot be converted into S locks. Instead, the S or IX lock is converted into an SIX lock.
- Stronger locks do not take precedence over weaker locks that already exist on a data object. A request for a new lock will not be granted until every existing, incompatible lock held by other transactions is released. It makes no difference whether the existing locks are stronger or weaker than the requested lock.
- Lock mode strength is used in the following ways by ALLBASE/SQL:
  - When several transactions are concurrently accessing the same data object, each has been granted a lock on the object. ALLBASE/SQL has computed which lock is the strongest, and has stored this information. When a new transaction requests a lock on the data object, or when one of the existing transactions tries to convert its existing lock to one that is stronger, the requested lock mode type is compared to the strongest existing lock on the data object. If the requested lock is compatible with the strongest lock, the request is granted; otherwise, the requesting transaction must wait.
  - A transaction does not need to acquire a new lock on a data object in either of the following situations:
    - The transaction already owns a lock on the data object, and the existing lock is the same or stronger than the lock that is needed.

      For example, if a transaction needs to read a row from a PUBLIC table, and it has already been granted an X lock on the page when it previously updated a different row on the page, the transaction does not need to obtain an S lock on the page. The existing X lock is stronger than the S lock that is needed.
    - The transaction already owns a lock at a higher granularity, and the existing lock is the same or stronger than the lock that is needed.

      For example, if a transaction has already been granted an SIX lock on a PUBLICROW or PUBLIC table, it does not need to acquire an S row or page lock when it reads a row. The SIX lock at the table level is stronger than the S lock that is needed at the lower level. But if the transaction writes a row, it does need to acquire an X row or page lock because the SIX lock at the table level is not strong enough.

Purpose of Slide

To demonstrate how the optimizer affects locking.

Understanding How the Optimizer Affects Locking

```
isql => GENPLAN FOR SELECT * FROM PurchDB.SupplyPrice
        WHERE VendPartNumber = '1010';

isql => SELECT * FROM SYSTEM.PLAN;
```

Query Block	Step	Level	Operation	Tablename	Owner	Indexname
1	1	1	Index Scan	SupplyPrice	PurchDB	VendPartIndex

Figure 4-27.

Key Points

- Locks on tables are acquired according to a combination of SQL statement, table type, and type of scan used to access the data. Remember that by default locks must be acquired before data can be read or written.
- When you issue an SQL statement, the optimizer chooses the type of scan that will be used to locate the rows that qualify for the query (it usually chooses a scan that will result in the fewest I/O operations needed to read the requested data). Four types of scans are possible.
- serial scan
- index scan
- hash scan
- TID scan

Each of these will be described in detail on the next few pages.

- The optimizer also makes decisions about join order, join method, and sort operations. The **access plan** for a query is the method chosen by the optimizer as the most efficient method to access the rows that qualify for the query.
- In ISQL, you can use the **GENPLAN** statement to generate the optimizer's access plan for a particular **SELECT**, **UPDATE**, or **DELETE** statement. The resulting access plan is inserted into the pseudo-table SYSTEM.PLAN. Please refer to the *ALLBASE/SQL Reference Manual* for more information about **GENPLAN** and SYSTEM.PLAN.

Purpose of Slide

To explain a serial scan.

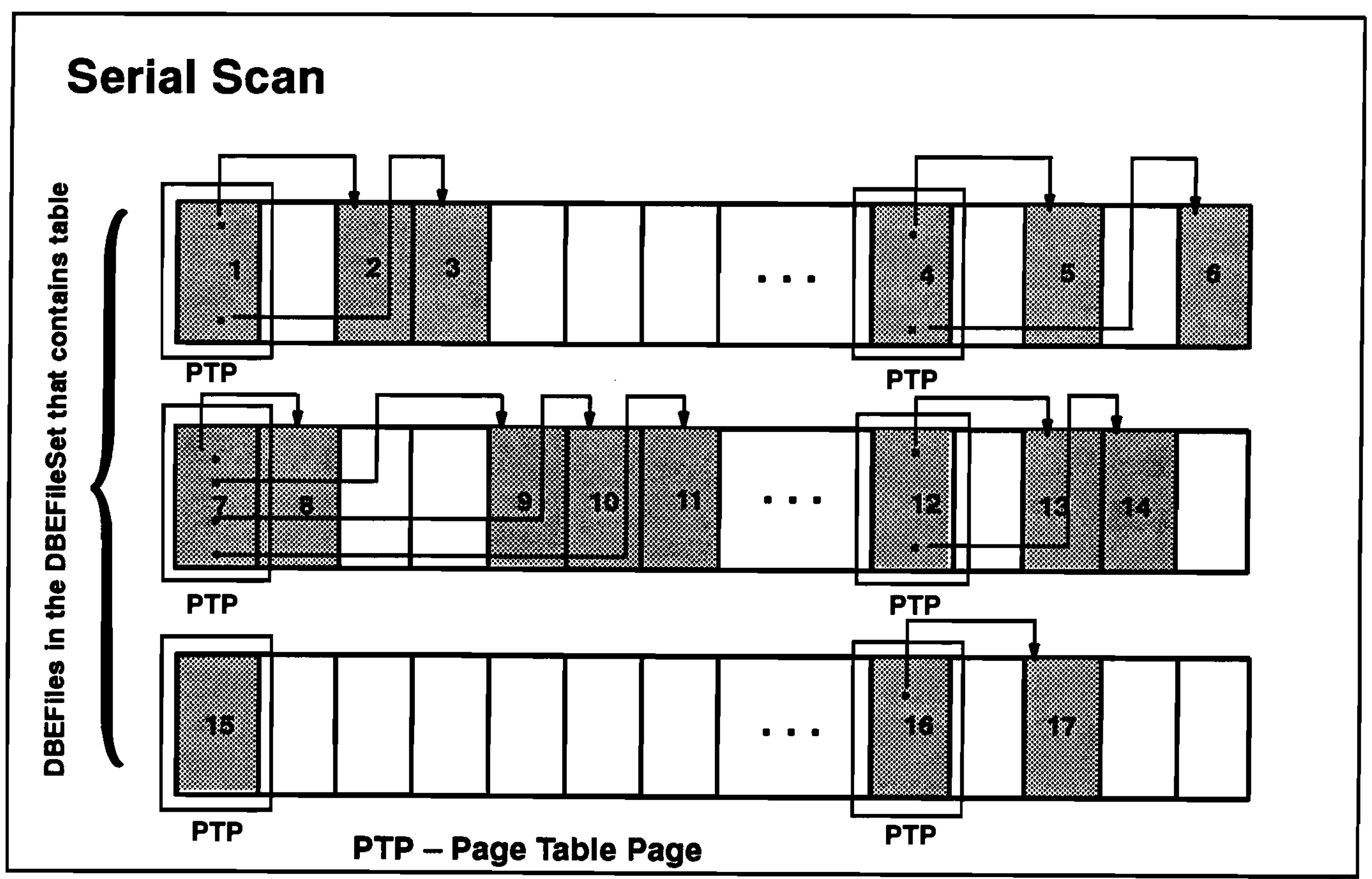


Figure 4-28.

Key Points

- Every DBEFile contains one or more **page table pages**, each of which is basically a table of contents for up to 252 of the pages. For every 253 pages in the DBEFILE, the first page is a page table page, and the other 252 pages are used to store table data or index data, depending on the type of the DBEFile. Each page table page consists of 252 entries: each entry consists of an object id that indicates which specific table or index has data stored on the page, and the physical address (also known as the TID) of the page.
- When a serial scan is performed over a table, every page table page is examined in each DBEFile in the DBEFileSet containing the table. If one or more entries exist for the table, the appropriate data page(s) are accessed directly by using the physical address stored on the page table page. Finally, all rows on each retrieved data page are examined to see if any qualify for the query.
- When a serial scan is performed over a table, all rows in the table are read in the order that they are physically stored.
- The I/O cost for a serial scan is equal to the number of page table pages that exist in the DBEFileSet, plus the number of pages that exist in the table.
- A serial scan usually requires more I/O than any other type of scan, unless all or most rows in the table will qualify for the query; in this case, a serial scan is actually the most efficient method for accessing the rows.
- When a serial scan is used, all of the rows in the table must be read to see if they qualify for the query. An S lock at the table level is the most efficient way of locking all rows for a read. An S lock at the table level on a PUBLICROW or PUBLIC table usually indicates a serial scan. Remember that an SIX lock is equivalent to an S lock plus an IX lock. An SIX lock at the table level can also indicate that a serial scan is being used.
  - When an index, hash, or TID scan is used, only some of the rows in the table must be locked. If the default isolation level is used (isolation levels will be discussed later in this module), then IS or IX locks at the table level indicate that something other than a serial scan is being used to locate rows in a PUBLICROW or PUBLIC table. If an isolation level other than the default is used, a serial scan will also generate IS or IX locks at the table level.
- If a serial scan is used to locate data in a PUBLICROW or PUBLIC table and the transaction uses the default isolation level, an S lock will be acquired at the table level during internal processing. If there is also a need to write to a row (or page), an IX lock at the table level is also required. Therefore, an SIX lock will also be internally obtained on the table:
  - If the need to write is known at the same time that the S lock is acquired (such as a `DECLARE CURSOR FOR UPDATE`), an SIX lock is specifically requested instead of an S lock.
  - If the need to write occurs after the S lock has been granted on the table (such as a `SELECT` without a WHERE clause, followed by an `UPDATE` statement), an internal request is made to upgrade the S lock to an SIX lock. In the case of a single transaction, S + IX = SIX, the upgrade can only occur if no other transaction holds an S lock on the table. If another S lock exists, the request for an SIX lock cannot be granted because it is not compatible with the S lock. In this case, the converting transaction will have to wait until the other transaction releases its lock.

Purpose of Slide

To describe an index scan.

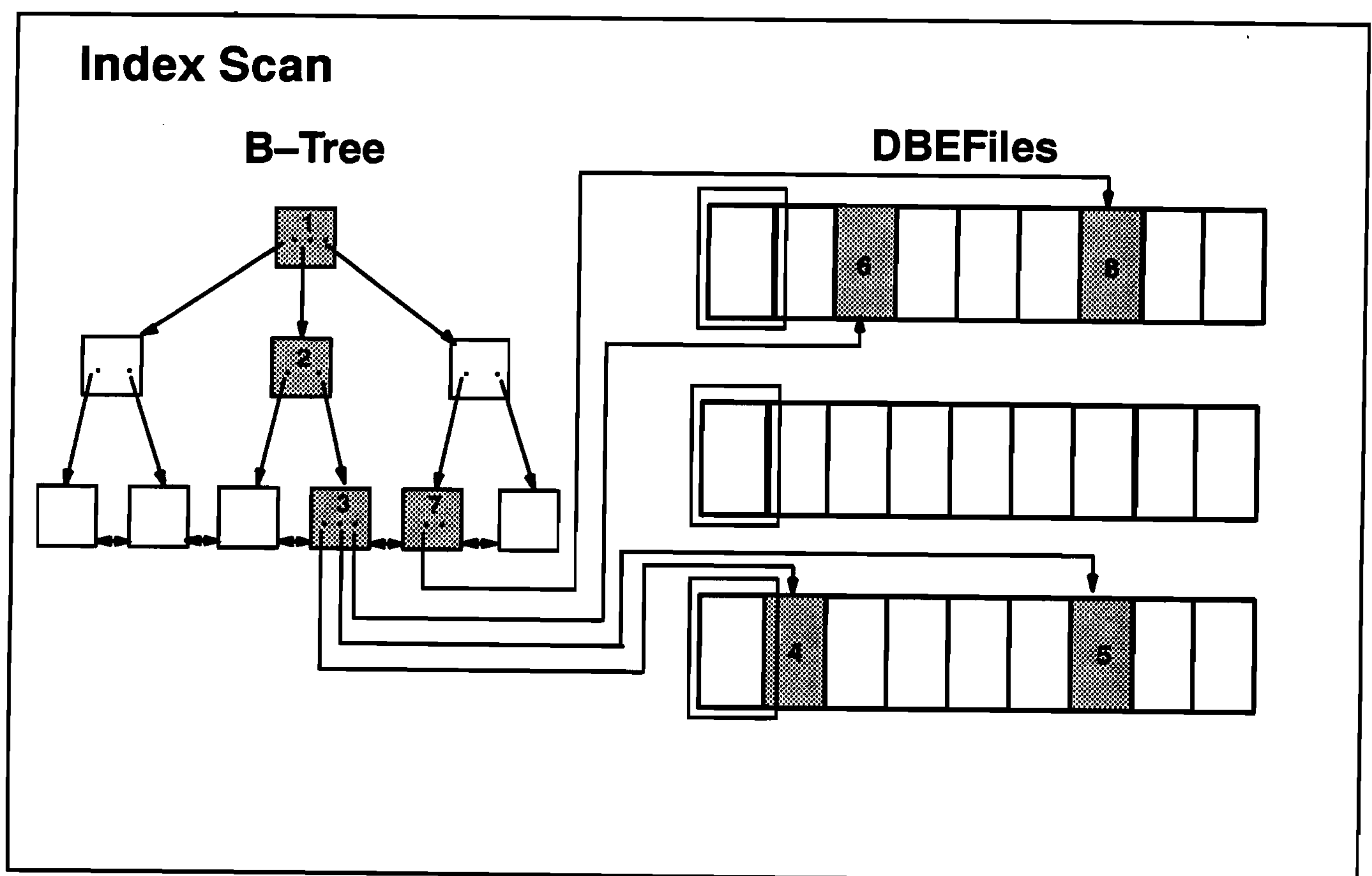


Figure 4-29.

Key Points

- At least one B-tree structure (index) must already exist on the table because the B-tree contains the pointers needed for the scan. A B-tree consists of a single root page and a number of leaf and non-leaf pages. Leaf pages are used to store index entries. Each leaf page consists of an index key value and the physical address of the row in the table which has that value in its key columns. A non-leaf page is used to store pointers to other index pages (either leaf pages or other non-leaf pages). Typically, the root page contains pointers to non-leaf pages (which may contain pointers to other non-leaf pages) that eventually contain pointers to leaf pages.
- When an index scan is performed over a table, a B-tree is traversed from the root page down to the appropriate leaf pages for index entries that have matching keys. Every matching index entry also contains the physical address of the row in the table, so the row can be accessed directly using the physical address if necessary. There are times when the row does not need to be accessed because the key values that are stored in the index itself are sufficient to satisfy the needs of the query.
- The I/O cost for an index scan is equal to the I/O that is needed to search the index, plus the I/O that is needed to access the rows that qualify for the query.
- When an index scan is used on a PUBLICROW or PUBLIC table, row (or page) level locking is used on the table.
- The optimizer might choose an index scan instead of a serial scan, even though all or most rows in the table will qualify for the query. This action is taken so that sorting can be avoided. The optimizer might choose an index scan when processing an ORDER BY, GROUP BY, DISTINCT, or UNION clause in a **SELECT** statement or when performing a sort/merge join. If all rows in a PUBLICROW or PUBLIC table qualify, many row or page locks will be acquired. Remember that if a query will read most or many rows in a table, it may be useful to use the **LOCK TABLE** command to acquire an S lock on the table to minimize deadlocks and shared memory needs.

Purpose of Slide

To describe a hash scan.

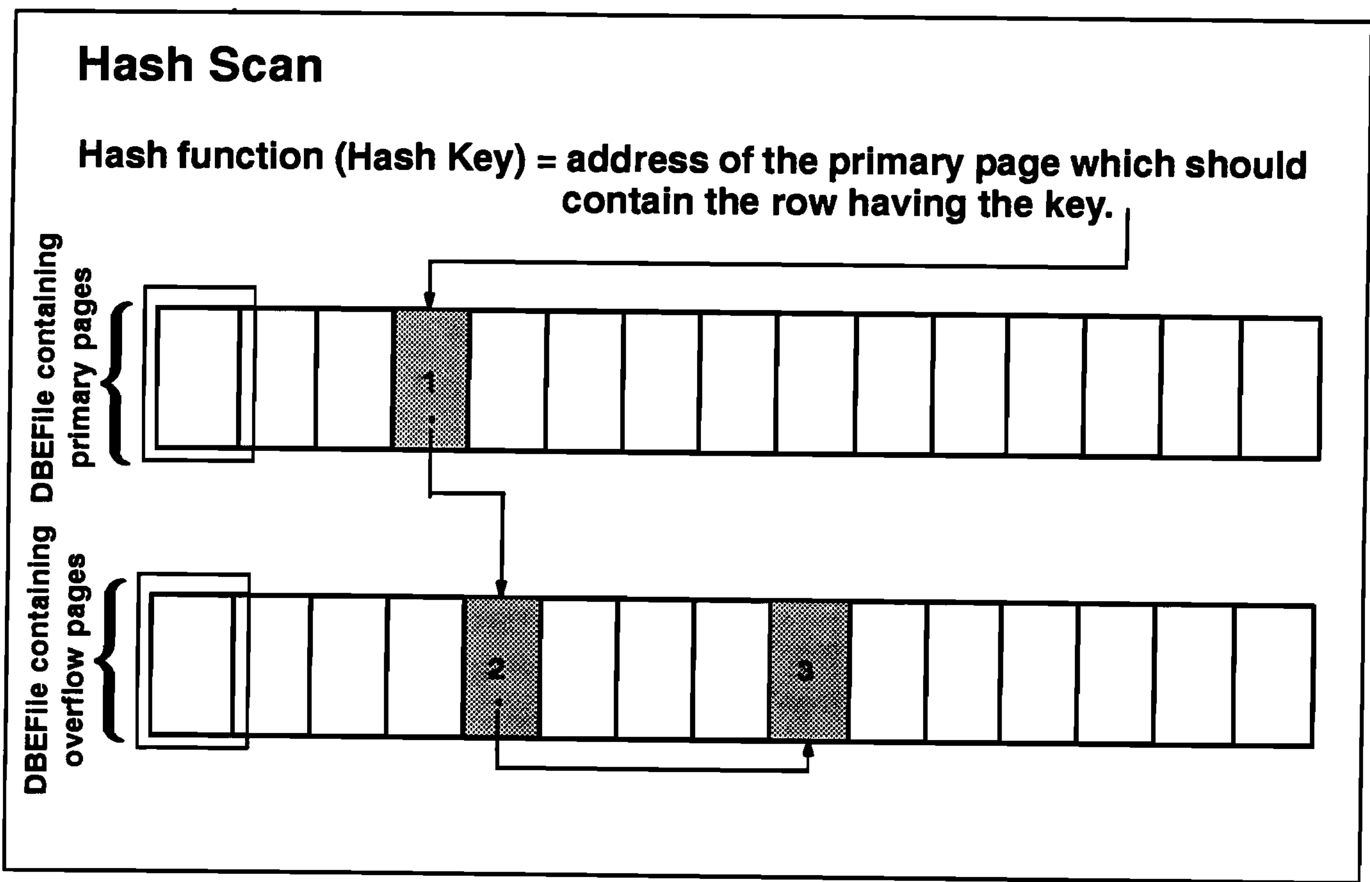


Figure 4-30.

Key Points

- A hash structure consists of primary pages and overflow pages. When a table is defined as hashed, a specific number of primary pages are allocated by ALLBASE/SQL for the storage of tuples in the table. When a row is inserted, an algorithm is used to calculate the physical address for the primary page on which the row should be stored. The algorithm uses the number of primary pages in the table and also the value of the **hash key** (that is, the value(s) in the key column(s) of the row) for the row that is being inserted. If space cannot be found on the primary page, an overflow page is allocated by ALLBASE/SQL and linked to the primary page; the row is then inserted onto the overflow page. An overflow page is only allocated if either
    - an overflow page does not already exist for the primary page, or
    - overflow pages exist, but they do not have space for the row.
- A hash scan can only be used to access a row in a table that has been created using a hash structure. Either the UNIQUE HASH ON clause or the HASH ON CONSTRAINT clause must have been used when the table was created (PUBLICROW tables cannot be created using a hash structure).
- A hash scan can only be used to locate a single row. However, you can issue a query in which multiple hash scans are executed to locate several rows (each scan returns a single row, and a union is performed over all of the scans).
- When a hash scan is performed, the algorithm that was described above is used to calculate the physical address for the primary page on which the row should be stored. A binary search is performed on this page for a row that matches the hash key. If the row is not found, a binary search is performed on each overflow page associated with the primary page until the row is found or all appropriate overflow pages have been examined.
- The maximum I/O cost for a hash scan is equal to one I/O for the primary page, plus one additional I/O for each overflow page associated with the primary page.
- A hash scan usually requires less I/O than an index scan, unless a serious overflow situation exists.
- When a hash scan is used to access a row in a PUBLIC table, page level locking is used.

Purpose of Slide

To describe a TID scan.

TID Scan

SELECT * FROM RecDB.CLUBS WHERE TID ( ) = 8:5:1;

TID = 8:5:1

1

Figure 4-31.

Key Points

- A TID scan can only be used to locate a single row. You can, however, issue a query in which multiple TID scans are executed to locate several rows (each scan returns a single row, and a union is performed over all of the scans).
- A row is accessed directly using the physical address (TID).
- The I/O cost for a TID scan is equal to one I/O.
- When a TID scan is used to access a row in a PUBLICROW or PUBLIC table, row (or page) level locking is used.
- When you use a TID function, you can assume that the optimizer will choose a TID scan. Please refer to the *ALLBASE/SQL Reference Manual* for more information about TID functions.

Purpose of Slide

To explain the locking of indexes.

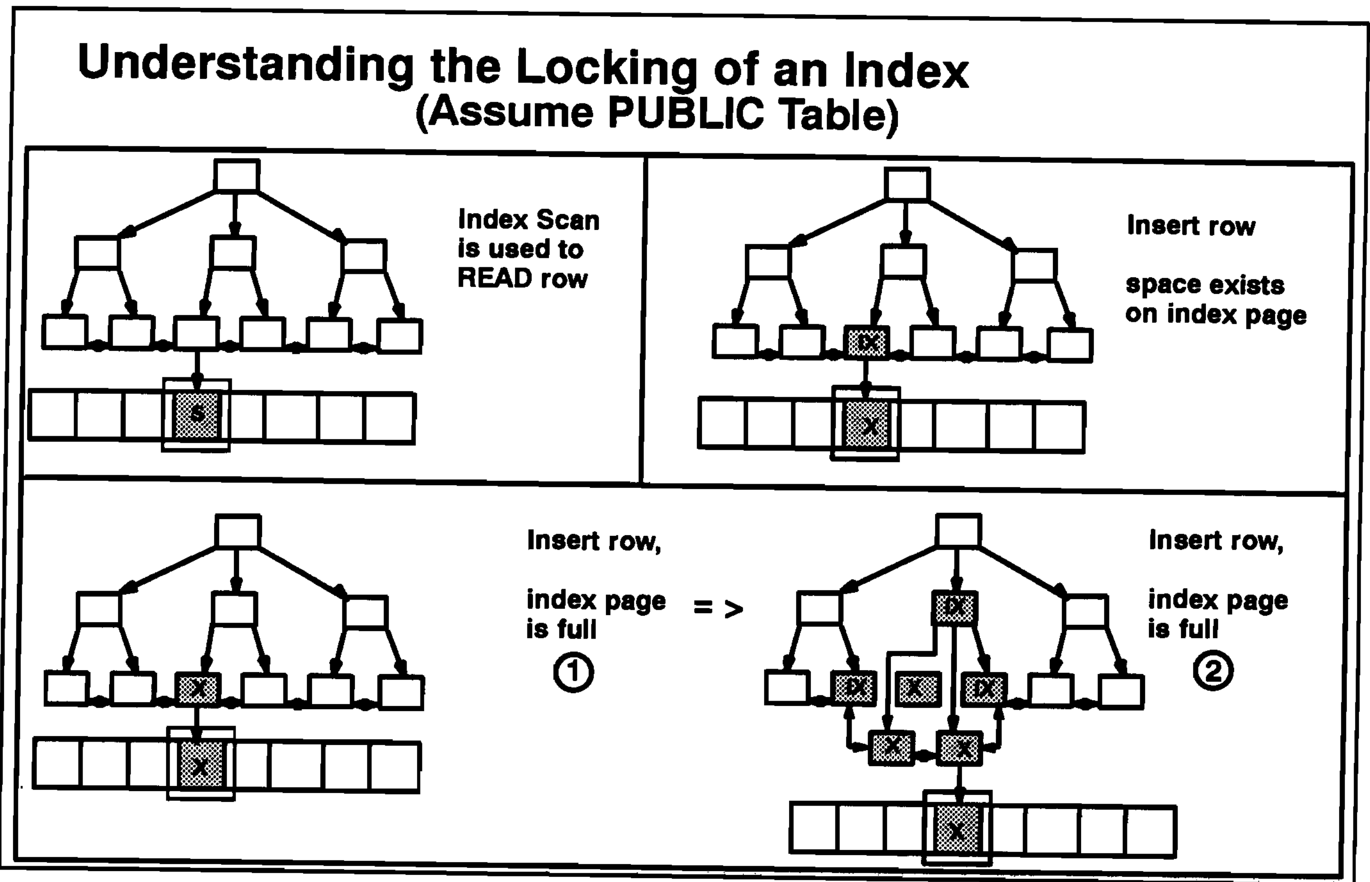


Figure 4-32.

Key Points

- Locks are never acquired on a B-tree index that has been defined on a PRIVATE or PUBLICREAD table. Concurrency control on the index is already achieved via the table level lock that is always acquired on the table. In other words, the S or X lock that is obtained on the table also covers the pages in the index.
- Locks are acquired in the following way on a B-tree index that has been defined on a PUBLICROW or PUBLIC table:
  - When an index scan is performed over the table, pages in the B-tree are not locked when they are traversed for index entries that contain matching keys. Only rows or pages in the table itself are locked when an index is used to read table data.
  - The index leaf page that covers the row in the table is locked with an IX lock when the index itself is modified, regardless of the type of scan used to modify the table.
  - If a row is inserted into the table, and space does not exist on the index page that should contain an index entry for the key, then the following occurs:
    - **START**—An X page lock is obtained on the index page, and it is examined to determine whether or not compression should be performed to create free space. If enough free space will be created, compression is performed and the index entry is inserted onto the page.
    - If compression would not result in enough free space for the index entry, it is not attempted (the page is full and must be split into two new pages). The data from the original page is moved to the new pages. At the end of the split operation, each of the new pages contains half of the index data from the original page, and the original page is freed.
    - The following locks exist on the index at the end of the split operation:
      - Three X page locks: one on the original leaf page, and one each on the two newly allocated leaf pages.
      - Usually three IX page locks: one on the parent (non-leaf) page of the original page, and one each on the two neighbor (leaf) pages of the original page (it is possible that there is only one neighbor, instead of two). The parent and neighbor page(s) must be modified to point to the newly allocated pages instead of to the original page. The neighbor page(s) must simply be updated. Both an update and an insert must occur for the parent page. There might not be enough space for the insert on the parent page, so the process beginning with the label **START:** can be repeated on each successive parent page until the root page of the index has been updated sucessfully.
  - If a row is deleted from the table, and an index page becomes empty because the last index entry on the page was deleted, ALLBASE/SQL frees the page. This requires an X page lock, regardless of the type of scan used to perform the delete.

Purpose of Slide

To demonstrate how cursors are used.

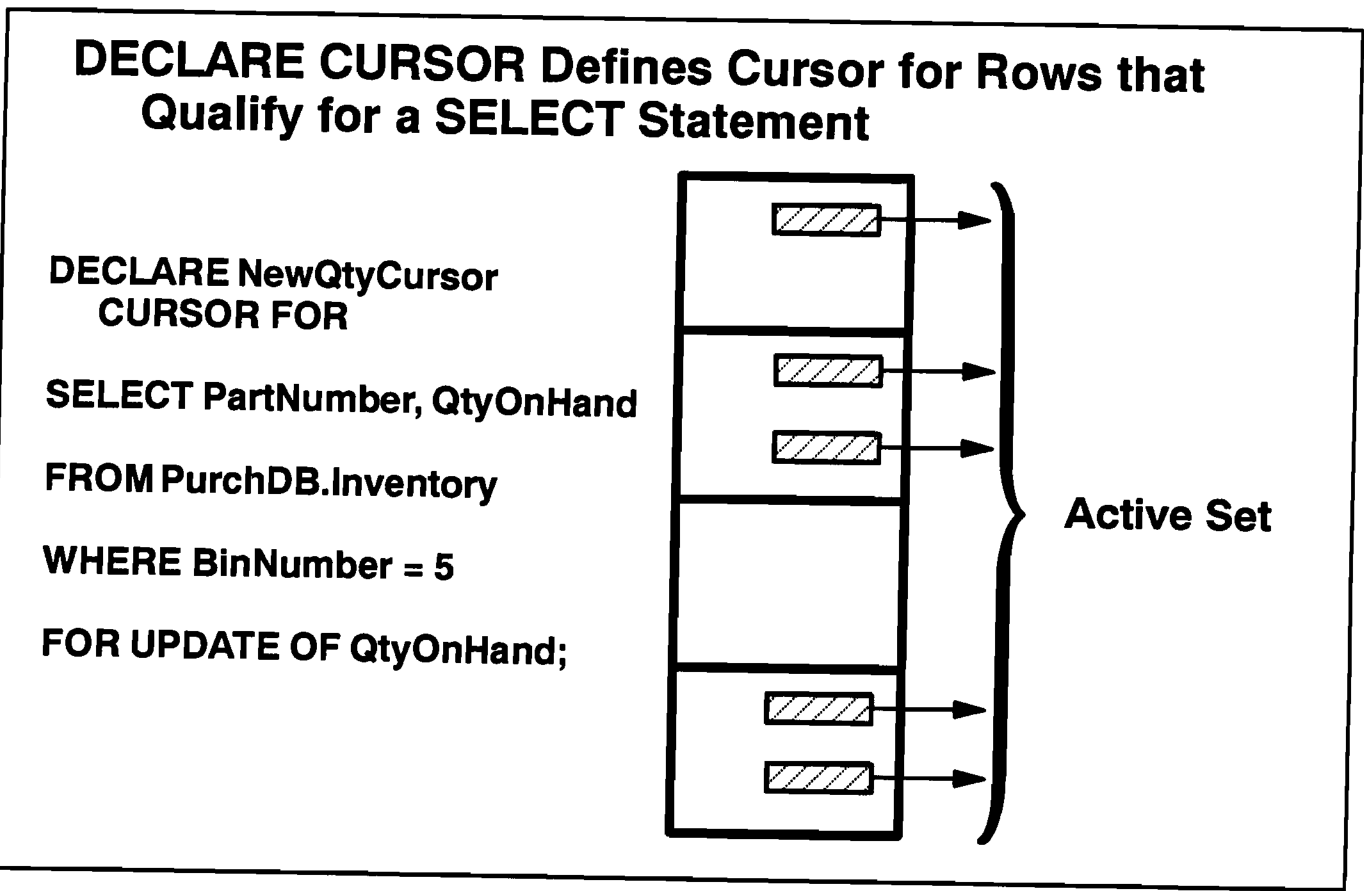


Figure 4-33.

- A cursor is a pointer that you advance through a set of rows retrieved with a SELECT statement. It can only be used within an application program. The primary SQL statements used to support cursors are:
  - DECLARE CURSOR
  - OPEN
  - FETCH
  - REFETCH
  - UPDATE WHERE CURRENT
  - DELETE WHERE CURRENT
  - CLOSE

Key Points

- The DECLARE CURSOR statement is used to associate a cursor with a specified SELECT statement:
  - The WHERE clause(s) of the SELECT statement determines the rows in the query result, which are also referred to as the active set.
  - When the DECLARE CURSOR statement is preprocessed, the optimizer normally computes the access plan for the rows in the active set, and ALLBASE/SQL stores this information as a section in the system catalog (a description of the section appears in the SYSTEM.SECTION view).
    - One of the ALLBASE/SQL preprocessors is normally used to preprocess SQL statements (including DECLARE CURSOR) prior to the compilation of an application program. During preprocessing, SQL statements are translated into compilable constructs that call ALLBASE/SQL external procedures at run time.
    - The DECLARE CURSOR statement cannot be issued interactively within ISQL. As a result, all other SQL statements that are used to support cursors are also not allowed within ISQL.
  - The DECLARE CURSOR statement supports an optional FOR UPDATE clause, which is used to specify the columns that might be updated when the cursor is used:
    - If you use the FOR UPDATE clause, the cursor must be an **updatable cursor**:
      - The cursor must be based on an updatable query. Generally, a query is updatable if it only involves one table (either directly or through a view), if it does not involve a sort operation, and if it is possible for ALLBASE/SQL to determine which particular rows and columns in the table should be modified. Please refer to the *ALLBASE/SQL Reference Manual* for more information about updatable queries.
      - The columns that are specified must actually correspond to columns in a base table. If a cursor is declared on a view, it is possible that some columns of the view are actually expressions or constants; these columns cannot be specified in the FOR UPDATE clause of a DECLARE CURSOR statement.
    - If you use the FOR UPDATE clause and the cursor is not updatable, an error will be returned when the DECLARE CURSOR statement is preprocessed. A cursor defined in a DECLARE statement that is not preprocessed successfully cannot be used in other SQL statements (OPEN, FETCH, REFETCH, UPDATE WHERE CURRENT, DELETE WHERE CURRENT, and CLOSE).

Purpose of Slide

To describe how cursors are used in ALLBASE/SQL statements.

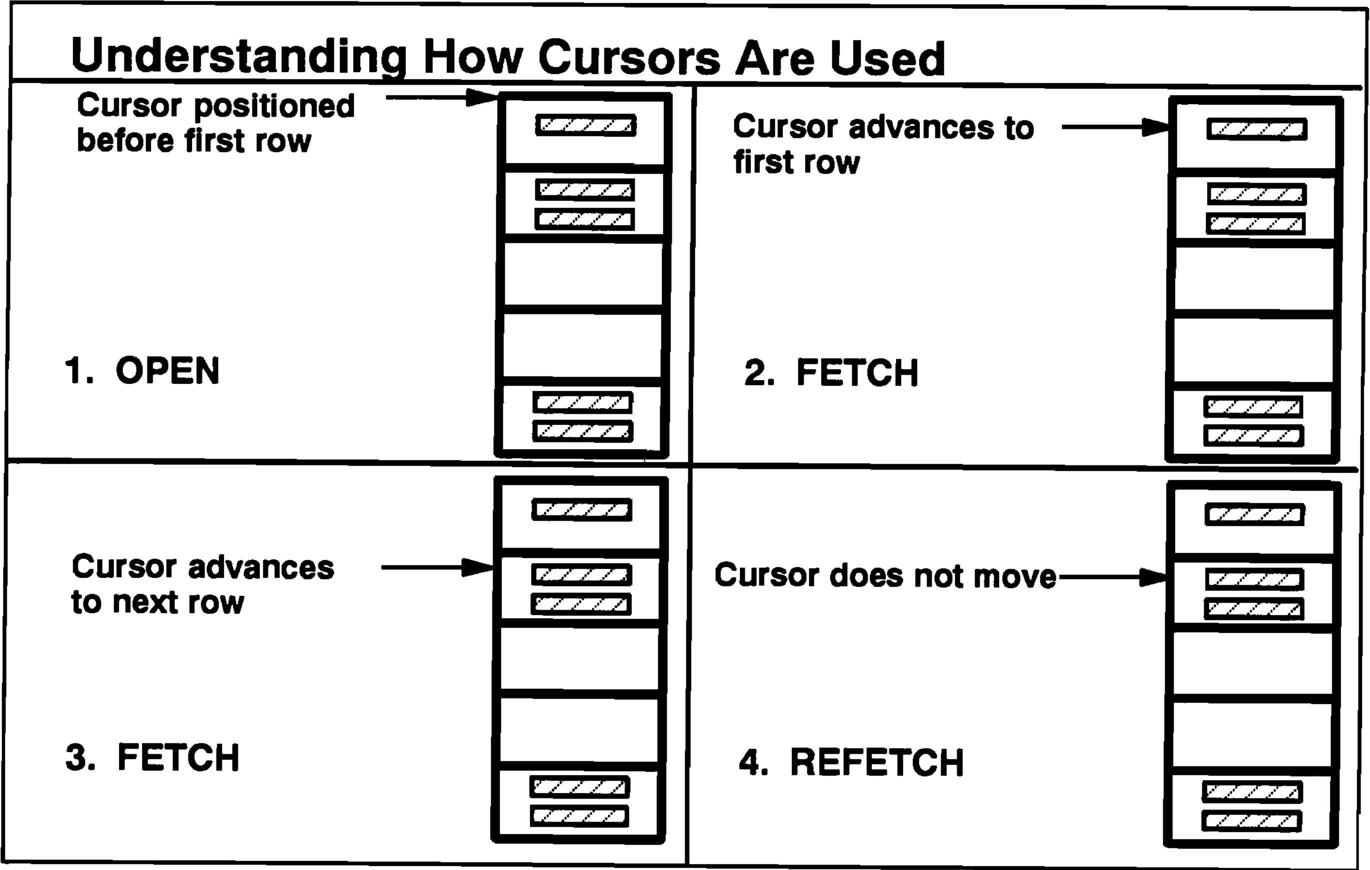


Figure 4-34.

Key Points

- The OPEN statement is used to begin execution of the stored section that was created by the DECLARE CURSOR statement. This execution results in opening appropriate scans to access rows in the active set. At the end of the OPEN statement, the cursor is positioned before the first row in the active set. A user may have more than one cursor open at the same time.
- The FETCH statement is used to move the cursor to the next row (or rows) in the active set and retrieve it (or them).
  - Generally, rows are locked when they are fetched.
  - An OPEN statement must be issued prior to the first FETCH statement.
  - The FETCH statement is normally used in a loop until all rows in the active set have been retrieved. The loop ends when a FETCH is made after the last row in the active set has been returned, and an error has been detected in the SQLCA.SQLCODE. At this moment, the cursor's position is undefined.
- The REFETCH statement is used to reacquire a lock on the row that is currently pointed to by the cursor. The cursor does not move. The REFETCH statement is only needed when a lock is not obtained or retained on a row when it is fetched (this situation will be described later in this module).
  - The REFETCH statement can only be used for a cursor that is updatable. An UPDATE WHERE CURRENT or DELETE WHERE CURRENT statement normally follows a REFETCH statement.
  - A FETCH statement must be issued prior to a REFETCH statement. If the FETCH fails (for example, if no more rows exist in the active set), the REFETCH statement cannot be used.
  - You only use the REFETCH statement to reaquire a lock on the last row that was fetched. REFETCH only operates on a single row.
- The UPDATE WHERE CURRENT and DELETE WHERE CURRENT statements can be issued to modify a single row immediately after it has been fetched or refetched (the cursor must still point to it):
  - If the FETCH (or REFETCH) fails, neither statement can be used. Do not use UPDATE WHERE CURRENT or DELETE WHERE CURRENT when you use FETCH to retrieve multiple rows (the BULK option).
  - If either statement is used, the cursor must be updatable.
  - If you use the UPDATE WHERE CURRENT statement, you may only update columns that were specified in the FOR UPDATE clause of the DECLARE CURSOR statement that was used to define the cursor.
- The CLOSE statement is used to close an open cursor. When a cursor is closed, its active set becomes undefined, and it can no longer be used in FETCH, UPDATE WHERE CURRENT, or DELETE WHERE CURRENT statements. To use a cursor again after it has been closed, you must issue another OPEN statement to reopen it.
- When a FETCH, REFETCH, UPDATE WHERE CURRENT, or DELETE WHERE CURRENT statement is preprocessed, ALLBASE/SQL stores a section for it in the system catalog.

Purpose of Slide

To describe how isolation levels are used.

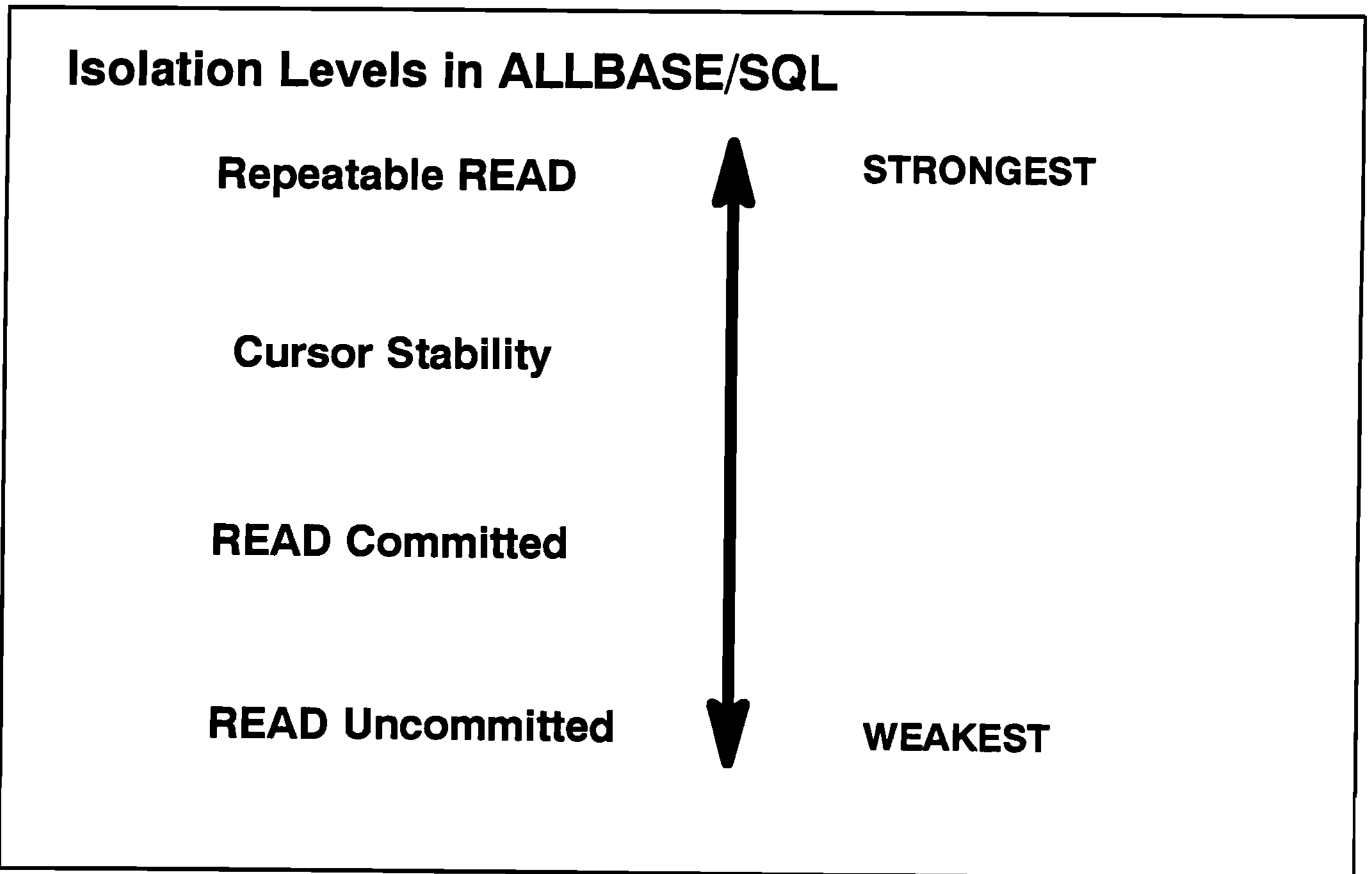


Figure 4-35.

Key Points

- When SQL statements that reference a user table are executed, locks of different kinds are obtained on the table by default. You can help control how long locks are held or if locks are obtained by altering the isolation level of the transaction.
- An isolation level is the degree to which a transaction is separated from all other concurrent transactions. In general, you should choose the least restrictive isolation level that meets each transaction's needs, to achieve the most concurrency.
- ALLBASE/SQL supports four isolation levels. They are briefly described below, in order from most restrictive to least restrictive. All of the isolation levels will be described in greater detail on the next few pages.
    - **Repeatable Read** (RR)—default.
        - All locks obtained by the transaction are held until the transaction ends.
    - **Cursor Stability** (CS)
        - S locks are released before the transaction ends (the application program has some control over when they are released).
    - **Read Committed** (RC)
        - S locks are automatically released by ALLBASE/SQL immediately after they are obtained.
    - **Read Uncommitted** (RU)
        - S locks are not acquired when the transaction reads data.
- The isolation level of a transaction is an attribute that can be specified using an optional clause in the `BEGIN WORK` statement. For example:

```
BEGIN WORK CS;
```

Purpose of Slide

To describe the Repeatable Read isolation level.

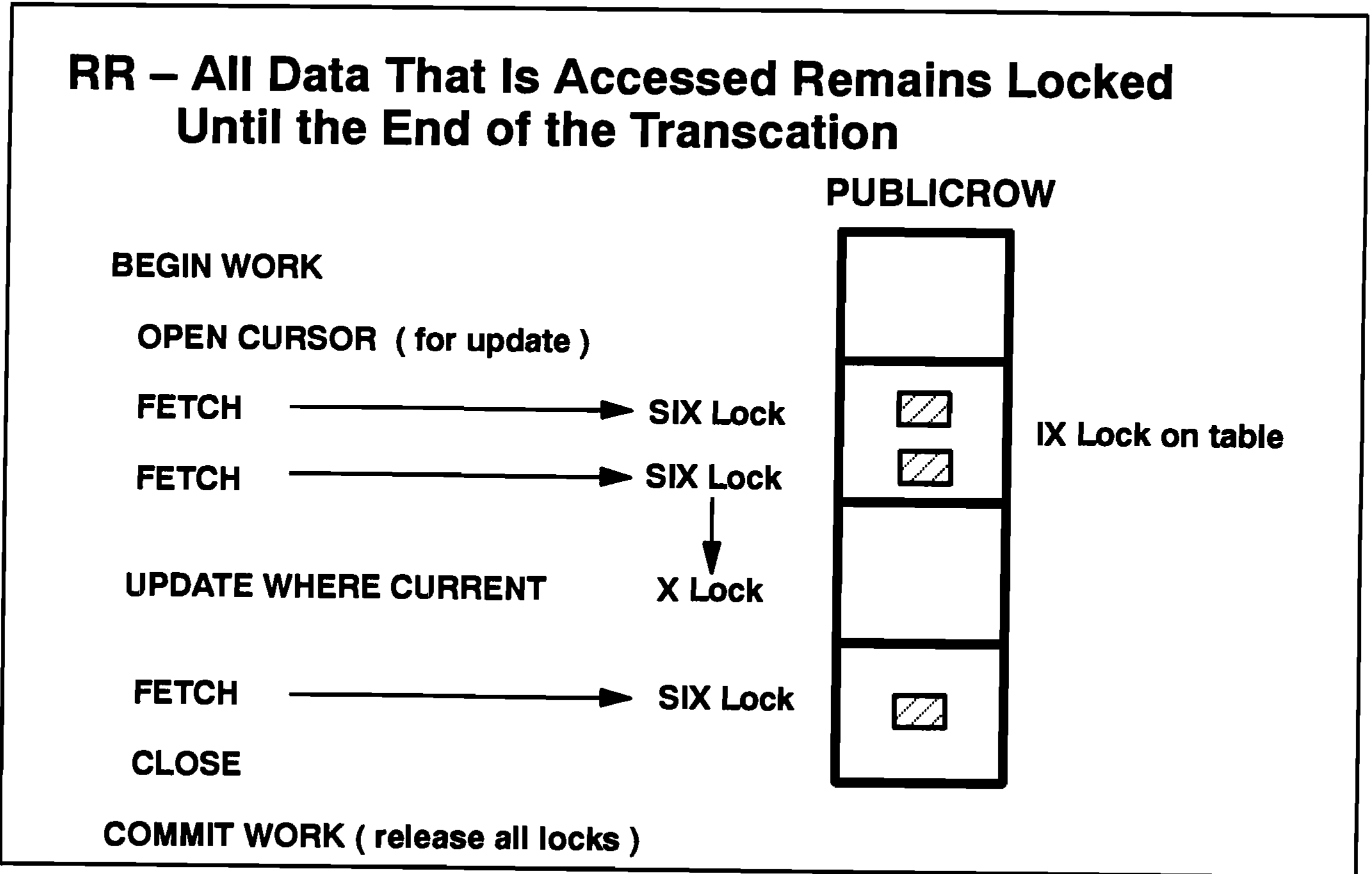


Figure 4-36.

Key Points

- **Repeatable Read** (RR)—All locks obtained by the transaction are held until the transaction ends.
- RR is the default isolation level in ALLBASE/SQL.
- Because S (or SIX) locks are obtained and retained on data that is read, the information cannot be modified by another transaction. If the transaction reads a row for a second time, the data in the row can only be different from the data that was read the first time if the transaction itself had modified the row. If the transaction had modified the row, the S or SIX lock would have been converted to an X lock.
- An RR transaction can read a row (that it has already read once) for a second (or third, etc.) time, and the information returned for the second read will be identical to the information that was returned for the first read. Hence, reads are repeatable.

Purpose of Slide

To describe the Repeatable Read isolation level (continued).

Summary of Locking in an RR Transaction

For UPDATE?	Type of Scan	Lock Type/Level
NO	Index/Hash/TID	S lock on row/page
NO	Serial	S lock on table
YES	Index/Hash/TID	SIX lock on row/page[1, 2, 3]
YES	Serial	SIX lock on table[1, 3]

[1] Reduces concurrency with other transactions that want to READ **all** of the data in the table.

[2] Reduces concurrency with other transactions that want to READ **some** of the data in the table.

[3] Reduces concurrency with other READ–FOR–UPDATE transactions.

Figure 4-37.

Key Points (continued)

- A cursor is a pointer that you advance through a set of rows associated with a SELECT statement. The FETCH statement is used to move the cursor and retrieve one or multiple rows in the active set.
- Assume that a cursor is used in an RR transaction. For each table involved in the query associated with the cursor, the following locks are obtained by default on rows in a PUBLICROW table or on pages in a PUBLIC table when rows are retrieved using the FETCH statement:
  - If the FOR UPDATE clause is not used in the DECLARE CURSOR statement, the rows are simply read:
    - If the optimizer chooses an index, hash, or TID scan to obtain the rows in the active set, S locks are obtained on the rows (or pages). Appropriate intention locks are also obtained.
    - If the optimizer chooses a serial scan to obtain the rows in the active set, a single S lock is obtained on the table.
    - If an S lock is obtained at the row, page, or table level, other transactions can read all rows (or pages) in the table that the DECLARE CURSOR transaction has read, but has not modified. In this case, concurrency with other read transactions is not reduced.
  - If the FOR UPDATE clause is used in the DECLARE CURSOR statement, some rows in a table are read with the intention of writing to them:
    - The access plan of any query can consist of a complex sequence of subqueries. If the access plan for the query on which the cursor is defined includes a subquery that simply reads data, locks for the data in that subquery will be obtained as described above. (In other words, S locks can be acquired by a cursor defined with a DECLARE CURSOR FOR UPDATE statement).
    - If the table is being updated, the following is true:
      - If the optimizer chooses an index, hash, or TID scan to access data in the table, SIX locks are obtained on the rows (or pages) that qualify. Appropriate intention locks are also obtained. When an SIX lock is obtained at the row or page level, other transactions cannot read any of the data that the SIX transaction has read but not modified.
      - If the optimizer chooses a serial scan to access data in the table, a single SIX lock is obtained on the table. Another transaction can read rows (or pages) in the table that the SIX transaction has read but not modified; however, the other transaction cannot read the rows with the intention of writing to them, and it cannot read all of the rows in the table.
      - If a transaction has obtained an SIX lock at the row, page, or table level, another transaction that wishes to read all of the rows in the table must wait until the SIX lock (and intention locks) are released.
    - In an RR transaction, SIX locks are held until the transaction terminates. Concurrency with other read or read with an intention to write transactions can be reduced when RR is used in transactions that contain updatable cursors.

Purpose of Slide

To describe the Repeatable Read isolation level (continued).

Locks Obtained By A Cursor in RR *

BEGIN WORK						
OPEN	↑					
FETCH	↑ S					
FETCH	S	↑ S				
FETCH	S	S	↑ S			
FETCH	S	S	S		↑ S	
FETCH	S	S	S		S	↑ S
CLOSE	S	S	S		S	S
COMMIT WORK						

* Assuming Index Scan and NON–UPDATABLE cursor

↑ – Cursor S – S row lock

Figure 4-38.

Key Points (continued)

- The example on the previous page shows the locks that are obtained by a non-updatable cursor in an RR transaction, if an index scan is chosen to access the rows in the active set.
  - If a serial scan is chosen instead of an index scan, a single S lock is obtained at the table level when the first fetch is made. Individual S locks at the row level are not obtained.
  - If the cursor is updatable and an index scan is chosen to access the rows in the active set, SIX row locks are obtained instead of S row locks.
  - If the cursor is updatable, and a serial scan is chosen, a single SIX lock is obtained at the table level when the first fetch is made.
- Use the RR isolation level for a transaction when you want to retain all locks until the transaction terminates. This is especially necessary when you need to repeatably read rows and be guaranteed that the rows have not changed from the first time that they were read.
- Use the RR isolation level for a transaction that contains a non-updatable cursor if you need to view a consistent snapshot of the data at a single point in time, and especially if you need to make data modifications based on the values returned by the cursor.
  - If you make data modifications, you must use SQL statements other than the `UPDATE WHERE CURRENT` or the `DELETE WHERE CURRENT` statements to accomplish your changes, because the data is not updatable through the cursor.
- Use the RR isolation level for a transaction in which you perform a `BULK FETCH` and then use the `UPDATE` statement (not `UPDATE WHERE CURRENT`) to modify the rows that were fetched.
  - Remember, do not use either the `UPDATE WHERE CURRENT` or the `DELETE WHERE CURRENT` statements when you perform a `BULK FETCH`. The cursor can only point to a single row at a time. When `FETCH` is used to retrieve multiple rows, the cursor sequentially moves to each row that is fetched during internal processing. At the end of the `BULK FETCH`, the cursor points to the last row that was returned. If an isolation level other than RR is used, the S or SIX locks that may have been obtained on the other rows in the `BULK FETCH` are released. To ensure data integrity (that is, to ensure that you do not accidentally overwrite changes made by another user), you must use the RR isolation level to retain locks on these rows to guarantee that they are not modified by another transaction before your `UPDATE` statement is executed.

Purpose of Slide

To explain the Cursor Stability isolation level.

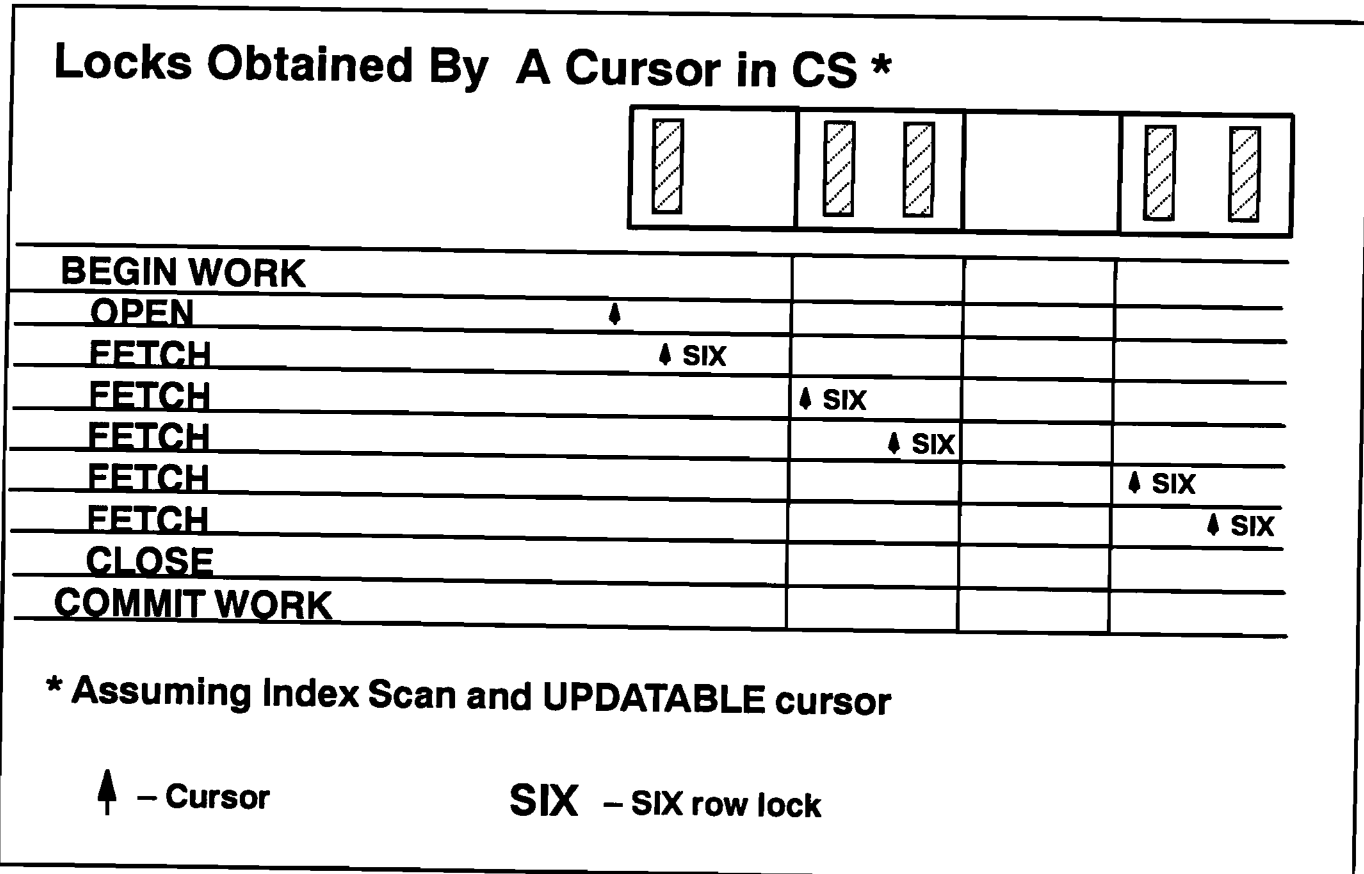


Figure 4-39.

Key Points

- **Cursor Stability** (CS)—S locks are released before the transaction ends (the application program has some control over when they are released).
- The primary use of Cursor Stability is to improve the concurrency of transactions that contain updatable cursors. When you use CS in a transaction with an updatable cursor on a PUBLICROW or PUBLIC TABLE, the following is true for that cursor:
  - During any `FETCH` statement, S or SIX locks might be obtained and released on rows (or pages) when locating the next row of the active set. Appropriate intention locks would also be obtained and released. In the example on the previous page, if a serial scan was used instead of an index scan, the following would occur to locate the first row in the active set:
    - The first row in the table would be locked with an SIX lock, and the row would be examined to see if it qualifies for the query associated with the cursor.
    - If it does not qualify, the SIX lock would be released, and the next row in the table would be locked with an SIX lock and examined.
    - This process repeats until a row is found that qualifies for the query. At the end of the the `FETCH`, only this last row remains locked with an SIX lock.
  - At the end of the `FETCH` statement, only a subset of rows in the query result remain locked:
    - The row (or page) that was just fetched has an SIX lock. This row is stable (that is, it cannot be changed by another transaction) as long as the cursor points to it (that is, until another `FETCH` statement is issued). An IX intention lock for this row also exists at the page level if the table is PUBLICROW.
    - If the CS transaction itself modifies this row, the existing SIX lock is converted to an X lock.
    - An IX intention lock exists at the table level, regardless of whether the optimizer has chosen a serial scan or an index, hash, or TID scan (using RR, an SIX lock is obtained on the table if a serial scan is chosen). An IX lock at the table level provides more concurrency than an SIX lock, especially to other similar transactions: IX is compatible with IX, but SIX is not compatible with SIX.
    - By default under any isolation level, X locks and IX intention locks on pages containing modified rows are retained until the transaction ends.
  - When the next `FETCH` is performed and the cursor moves, the following locks are released:
    - SIX locks held on rows (or pages) that were not updated.
    - IX page locks for pages on which rows were not modified. If the cursor moves to another row on the same page as the last row that was fetched, the IX page lock is retained (instead of being released and reacquired).

Purpose of Slide

To explain the Cursor Stability isolation level (continued).

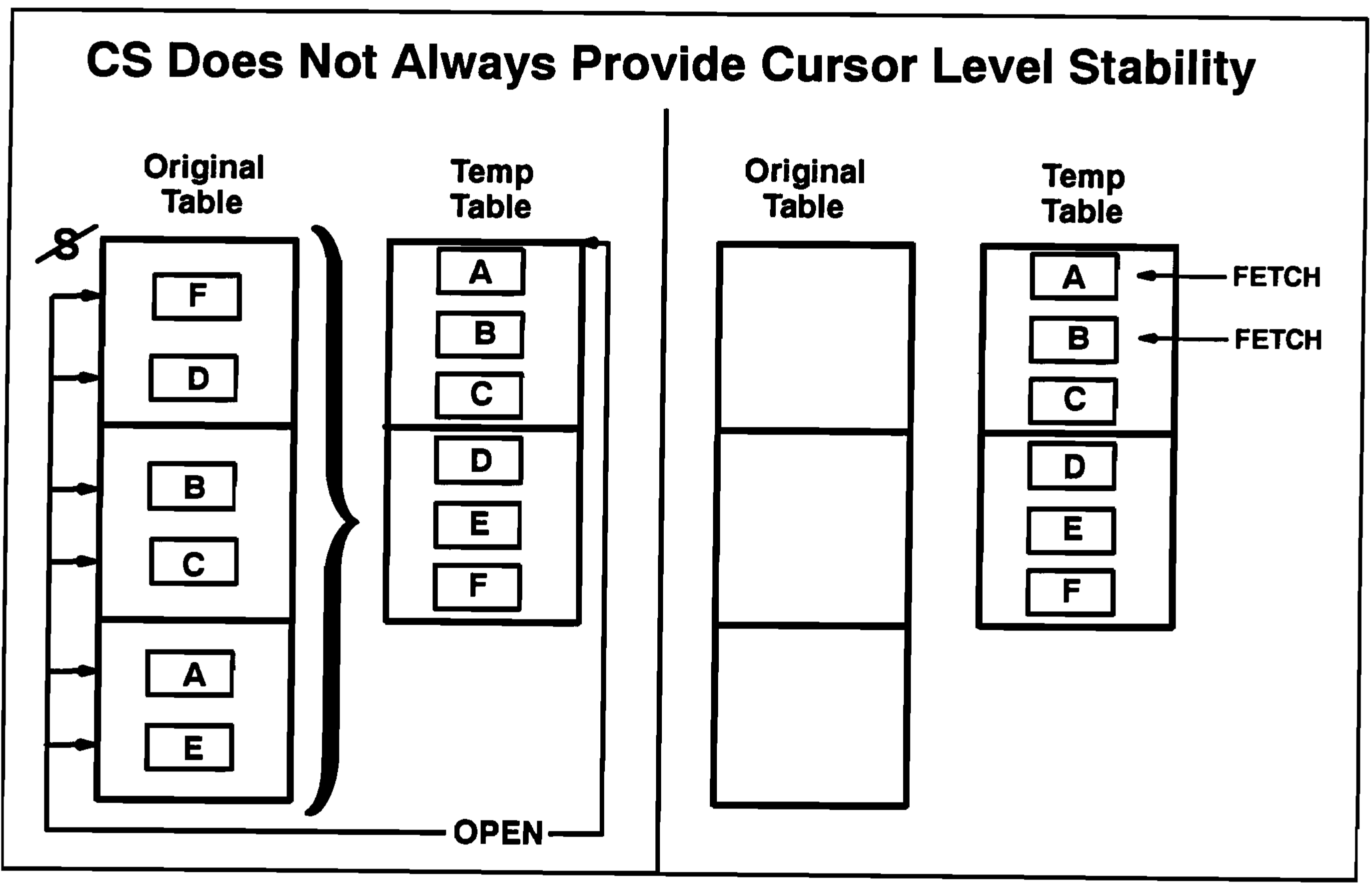


Figure 4-40.

Key Points (continued)

- If CS is used in a transaction having a non-updatable cursor, cursor level stability is not guaranteed. For example, if a temporary table is used to access the rows in the active set, the following is true:
  - A query that involves a sort operation (such as an ORDER BY, GROUP BY, DISTINCT, or UNION, or a query that uses a sort/merge join to join tables) may use a temporary table for the query result. When CS is used in such cases, your cursor actually points to rows in this temporary table, not to rows in a user table.
  - The temporary table is created when the `OPEN` statement is issued.
  - When CS is used, S locks are obtained and released on the user tables from which data is retrieved when the temporary table is created. Pages that might appear to be accessed by the current transaction's cursor are actually not locked at all, and other transactions are able to modify these pages.
  - When you issue each `FETCH` statement, ALLBASE/SQL does not re-sort to create a new temporary table, it simply retrieves another row from the existing temporary table. If your transaction does not retain locks on the original user table, other transactions can modify it (or even drop it). Therefore, your transaction may fetch a row that logically does not exist any more, or it may see an older version of a row that has since been modified. In such cases, it is the application developer's responsibility to maintain data integrity by verifying the current value of a row before updating it or using it as the basis for updating another table.
  - To retain S or SIX locks on a user table in a transaction that includes a non-updatable cursor, use one of the following:
    - Use the RR isolation level.
    - Use the `LOCK TABLE` command on the table at the start of the transaction. A table level lock will be obtained and retained until the transaction ends.
- CS provides greater read and write concurrency than RR to other transactions on data read by updatable cursors:
  - Greater read concurrency is achieved because other users can read rows as soon as a CS transaction moves the cursor. In an RR transaction, users must wait until the transaction terminates if they need to read rows (or pages) that have obtained SIX locks by the RR transaction.
  - Greater write concurrency is achieved because other users can modify rows as soon as a CS transaction moves the cursor. Users must wait until the transaction terminates if they need to modify rows (or pages) that have obtained S or SIX locks by the RR transaction.
- Use the CS isolation level for transactions that contain updatable cursors that need to scan through rows of committed data in a table, but may only update a few. If the cursor is updatable, then CS guarantees that a row will not change between the time you issue the `FETCH` statement and the time you issue an `UPDATE WHERE CURRENT` statement in the same transaction.

Purpose of Slide

To describe the Read Committed isolation level.

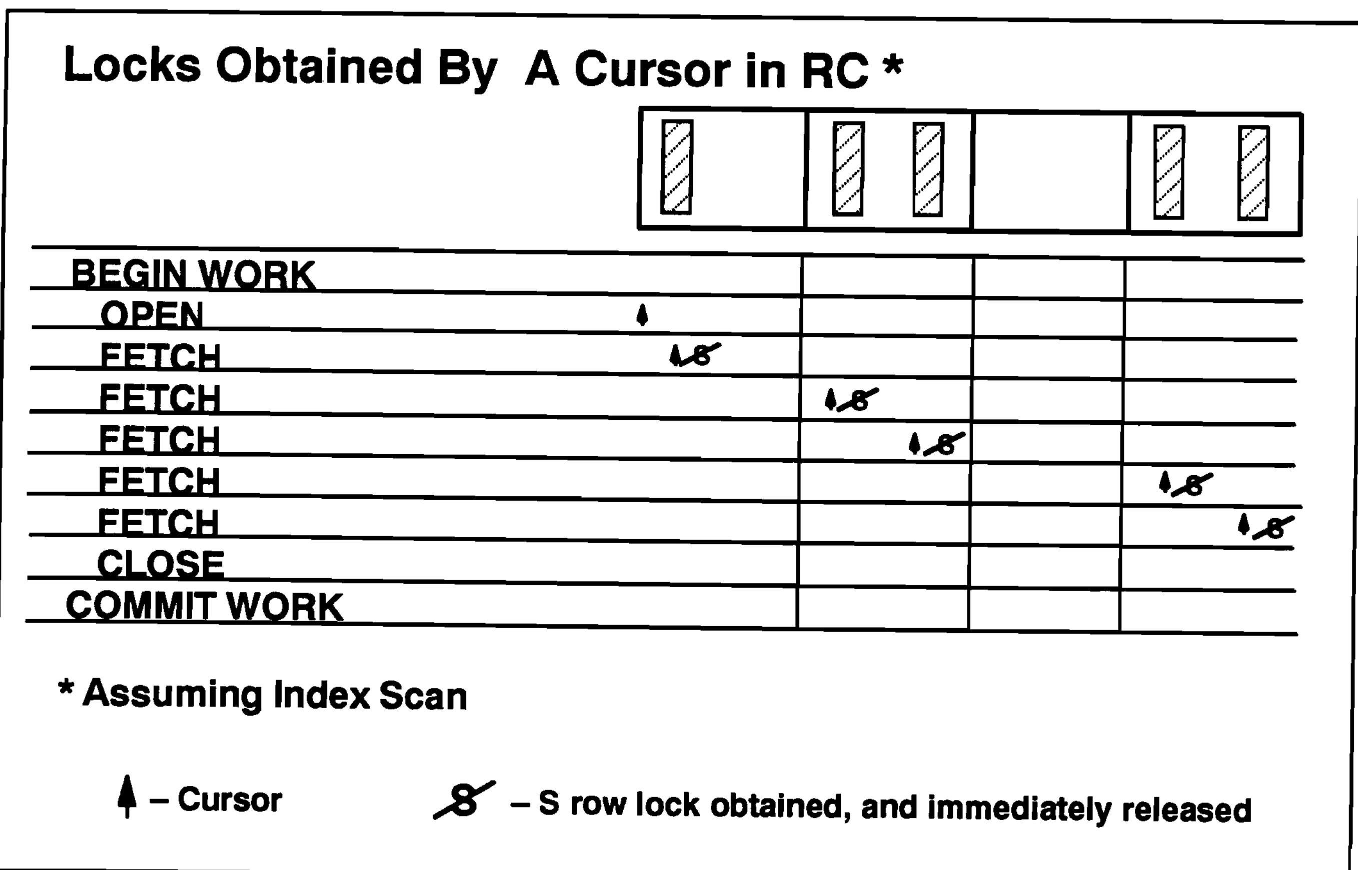


Figure 4-41.

Key Points

- **Read Committed** (RC)—S locks are automatically released by ALLBASE/SQL immediately after they are obtained.
- RC has the same characteristics as CS, except that locks are released immediately after each read, instead of waiting until the cursor moves (that is, until the next FETCH is issued). ALLBASE/SQL acquires (and releases) appropriate S locks (and intention locks) when each FETCH statement is issued. S locks are always obtained for both updatable and non-updatable cursors (SIX locks are not obtained and released for updatable cursors).
- During any FETCH statement, S locks might be obtained and released on rows (or pages) when locating the next row of the active set. Appropriate intention locks would also be obtained and released (the internal processing is similar to CS processing). At the end of the FETCH, the retrieved row is not locked and none of the other examined rows are locked.
- With RC, as with RR and CS, you can only retrieve rows that have been committed by some other transaction. In other words, you cannot read rows (or pages) that have been modified (locked with an X lock) or are in the process of being modified (locked with an SIX lock) by some other transaction that has not yet terminated.
- If you need to update or delete a row using a cursor in an RC transaction, you must use the REFETCH statement to verify that the row has not changed between the FETCH statement and the UPDATE WHERE CURRENT or the DELETE WHERE CURRENT statement. The S lock is released immediately after the FETCH, so another transaction may have modified the row after the FETCH was issued. To ensure that your transaction does not accidentally overwrite changes made by some other transaction, use the REFETCH statement to retrieve the row for a second time and examine its current value before updating or deleting it. A row cannot be changed by another transaction between the time you issue the REFETCH statement and the time you issue an UPDATE WHERE CURRENT statement in the same transaction, because the REFETCH statement obtains an SIX lock (at the row level on a PUBLICROW table and at the page level on a PUBLIC table).
    - If another transaction has modified (or is modifying) the row (or page) but has not yet committed the change (that is, has acquired an X or SIX lock on the data), the REFETCH transaction will wait (if another transaction is reading the data, the REFETCH transaction will also wait). After the SIX lock has been granted, the REFETCH transaction can determine whether or not other changes were made to the row between the time of the FETCH and the time of the REFETCH, and then modify the row appropriately.
- The UPDATE WHERE CURRENT and DELETE WHERE CURRENT statements will obtain an X lock at the row level on a PUBLICROW table and at the page level on a PUBLIC table. Appropriate intention locks are also acquired.
- By default under any isolation level, X locks and IX intention locks on pages containing modified rows are retained until the transaction ends. SIX locks obtained by the REFETCH command are also retained until the transaction ends, if they are not converted to X locks.

Purpose of Slide

To compare RC to the RR and CS Isolation Levels.

Comparing RR, CS, RC *

	RR	CS	RC
BEGIN WORK			
OPEN	↑	↑	↑
FETCH	↑ SIX	↑ SIX	↑ ~~S~~
FETCH	SIX ↑ SIX	↑ SIX	↑ ~~S~~
FETCH	SIX SIX ↑ SIX	↑ SIX	↑ ~~S~~
CLOSE	SIX SIX SIX		
COMMIT WORK			

* Assuming Index Scan and UPDATABLE cursor

↑ – Cursor

SIX – SIX row lock

~~S~~ – S row lock obtained, and immediately released

Figure 4-42.

Key Points (continued)

- Compared to RR, RC provides greater read and write concurrency to other transactions:
  - Greater read concurrency is achieved because other users can read rows immediately after an RC transaction has read them. In an RR transaction, users must wait until the transaction terminates before they can read rows (or pages) that have obtained SIX locks by the RR transaction.
  - Greater write concurrency is achieved because other users can modify rows immediately after an RC transaction has read them. In an RR transaction, users must wait until the transaction terminates before they can modify rows (or pages) that have obtained S or SIX locks by the RR transaction.
- Compared to CS, RC also provides greater read and write concurrency to other transactions:
  - Greater read concurrency is achieved because users can read rows immediately after an RC transaction has read them. In a CS transaction, users must wait until the transaction moves the cursor, before they can read rows (or pages) that have obtained SIX locks by the CS transaction.
  - Greater write concurrency is achieved because other users can modify rows immediately after an RC transaction has read them. In a CS transaction, users must wait until the transaction moves the cursor, before they can modify rows (or pages) that have obtained S or SIX locks by the CS transaction.
  - RC automatically acquires and releases intention locks during every FETCH. CS only releases appropriate intention locks when the cursor moves. In a CS transaction, the table level intention lock is always retained. If the cursor stays on the same page when the next FETCH is issued, the page level intention lock is also retained (instead of being released and reacquired). As a result, CS may incur less lock management overhead (and therefore be more efficient) than RC. Of course, some concurrency might be lost by using CS instead of RC in this situation.
- Use the RC isolation level for transactions that contain non-updatable cursors if you simply need to view a snapshot of committed data and if you don't need to make data modifications based on the values returned by the cursor.
- Use the RC isolation level for transactions that contain updatable cursors that need to scan through rows of committed data in a table, especially when a relatively large amount of time elapses between fetches. Since RC does not guarantee that a row will not change between the time you issue the `FETCH` statement and the time you issue an `UPDATE WHERE CURRENT` or a `DELETE WHERE CURRENT` statement, you must issue a `REFETCH` statement and examine the row's current values before you make any changes to it.

Purpose of Slide

To describe the Read Uncommitted isolation level.

Figure 4-43.

Key Points

- **Read Uncommitted** (RU)—S locks are not acquired by the transaction when it reads data.
- An RU transaction does not have to wait for an incompatible lock to be released by another transaction before it can read data. You can read rows (or pages) that have been modified (locked with an X lock), or are being modified (locked with an SIX lock) by some other transaction that has not yet committed its changes. The other transaction may eventually decide to issue a `ROLLBACK WORK` and cancel its changes.
- Reads made in an RU transaction are also known as dirty reads.
- If you need to update or delete a row using a cursor in an RU transaction, you must use the `REFETCH` statement to verify that the row has not changed between the `FETCH` statement and the `UPDATE WHERE CURRENT` or the `DELETE WHERE CURRENT` statement. After the `REFETCH`, examine the row's current value before updating or deleting it. This action ensures that your transaction does not accidentally overwrite changes made by some other transaction. A row cannot be changed by another transaction between the time you issue the `REFETCH` statement and the time you issue an `UPDATE WHERE CURRENT` statement in the same transaction, because the `REFETCH` statement obtains an SIX lock.
- The `UPDATE WHERE CURRENT` and `DELETE WHERE CURRENT` statements will obtain an X lock at the row level on a PUBLICROW table and at the page level on a PUBLIC table. Appropriate intention locks are also acquired.
- By default under any isolation level, X locks and IX intention locks on pages containing modified rows are retained until the transaction ends. SIX locks obtained by the `REFETCH` command are also retained until the transaction ends, if they are not converted to X locks.

Purpose of Slide

To compare RU to the RR, CS and RC Isolation Levels.

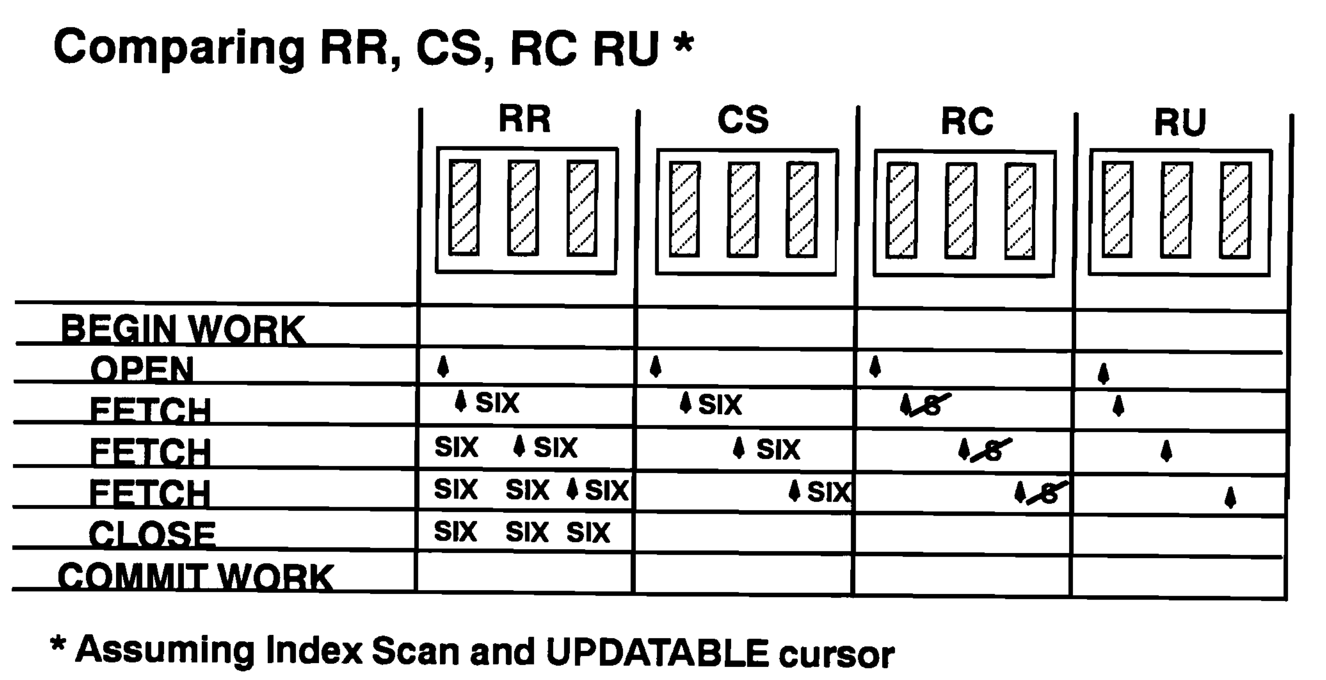


Figure 4-44.

Key Points

- Compared to RR, RU provides greater read and write concurrency to other transactions:
  - Greater read concurrency is achieved because other users can immediately read rows that the RU transaction has read. In an RR transaction, users must wait until the transaction terminates before they can read rows (or pages) that have obtained SIX locks by the RR transaction.
  - Greater write concurrency is achieved because other users can immediately modify rows that the RU transaction has read. In an RR transaction, users must wait until the transaction terminates before they can modify rows (or pages) that have obtained S or SIX locks by the RR transaction.
- Compared to CS, RU also provides greater read and write concurrency to other transactions:
  - Greater read concurrency is achieved because other users can immediately read rows that the RU transaction has read. In a CS transactions, users must wait until the transaction moves the cursor before they can read rows (or pages) that have obtained SIX locks by the CS transaction.
  - Greater write concurrency is achieved because other users can immediately modify rows that the RU transaction has read. In a CS transaction, users must wait until the transaction moves the cursor before they can modify rows (or pages) that have obtained S or SIX locks by the CS transaction.
- RU and RC both provide the same amount of read concurrency to other transactions. The write concurrency of other transactions might be slightly better using RU instead of RC.
  - The same read concurrency is achieved because other users can immediately read rows that either an RC or an RU transaction has read. An RU transaction does provide higher read concurrency to itself, though, because it does not have to wait to acquire an S lock before it can read data.
  - Slightly greater write concurrency might be achieved using RU because other users can immediately modify rows that the RU transaction has read. In an RC transaction, users must wait until the transaction has released its S locks (ALLBASE/SQL releases these locks immediately, but some overhead is still involved that could technically slow other write transactions).
- RU is ideal for *fuzzy* reports and similar applications where the reading of uncommitted data is not a major concern. Since RU does not guarantee that a row will not change between the time you issue the `FETCH` statement and the time you issue an `UPDATE WHERE CURRENT` or a `DELETE WHERE CURRENT` statement, you must issue a `REFETCH` statement and examine the row's current values before you make any changes to it.
- RU is the most efficient isolation level in ALLBASE/SQL. If you have data that only has readers (that is, you know in advance that no user will ever modify it), then for optimal performance you can use RU in a transaction that only accesses this data, because concurrency control by ALLBASE/SQL is not really necessary. Keep in mind, though, that if writers are ever acquired on the data, the isolation level of these RU transactions may need to be changed to RC.

Purpose of Slide

To describe when to use each isolation level (updatable cursor).

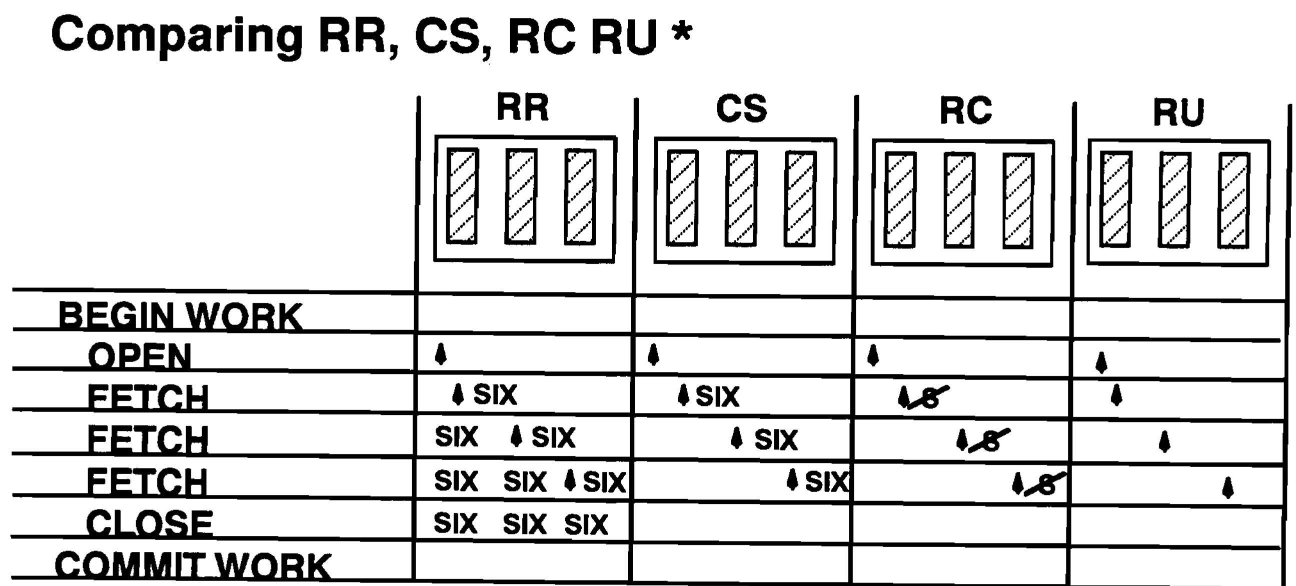


Figure 4-45.

Key Points

- If the transaction you are using has an **updatable cursor**, the following can help you select the appropriate isolation level.
  - If the transaction needs to retain all locks until the end of the transaction (it needs to repeatably read rows and be guaranteed that the rows have not changed from the first time that they were read), use the RR isolation level.
  - If the transaction uses a `BULK FETCH` statement to manipulate the cursor and uses the `UPDATE` statement to modify the rows that were fetched, use the RR isolation level.
  - If the transaction needs to scan through rows of committed data in a table and it will update all or most of the rows in the active set, use the CS isolation level:
    - If all of the rows in the table are in the active set and all rows are updated, CS will behave like RR because all row (or page) locks will be X locks, and X locks are held until the end of the transaction.
    - If the active set does not include all rows in the table, or if some rows (or pages) are not updated, CS is better than RR because some S or SIX locks will be released prior to the end of the CS transaction.
    - If all or most of the rows in the table are in the active set, you may also want to issue the `LOCK TABLE IN EXCLUSIVE MODE` statement to minimize shared memory needs.
  - If the transaction needs to scan through rows of committed data in a table, but it will only update a few of the rows in the active set, use either of the following isolation levels.
    - Use the RC isolation level for maximum concurrency, especially when
      - A relatively large amount of time elapses between fetches and updates, or
      - The number of users that will access the table concurrently is large (the cost of holding an SIX lock is high).
    - Use the CS isolation level for maximum lock efficiency in ALLBASE/SQL, especially when either of the following situations is true.
      - A small amount of time elapses between fetches and updates.
      - The number of users that will access the table concurrently is small (the cost of holding an SIX lock is low). If you use CS, there is no need to use the `REFETCH` statement. This improves performance and also reduces the number of sections in the DBEnvironment.
      - If the transaction can scan through rows of uncommitted data in addition to rows of committed data, use the RU isolation level to update a few of the rows. Use CS if the transaction will update all or most of the rows.

Purpose of Slide

To describe when to use each isolation level (non-updatable cursor).

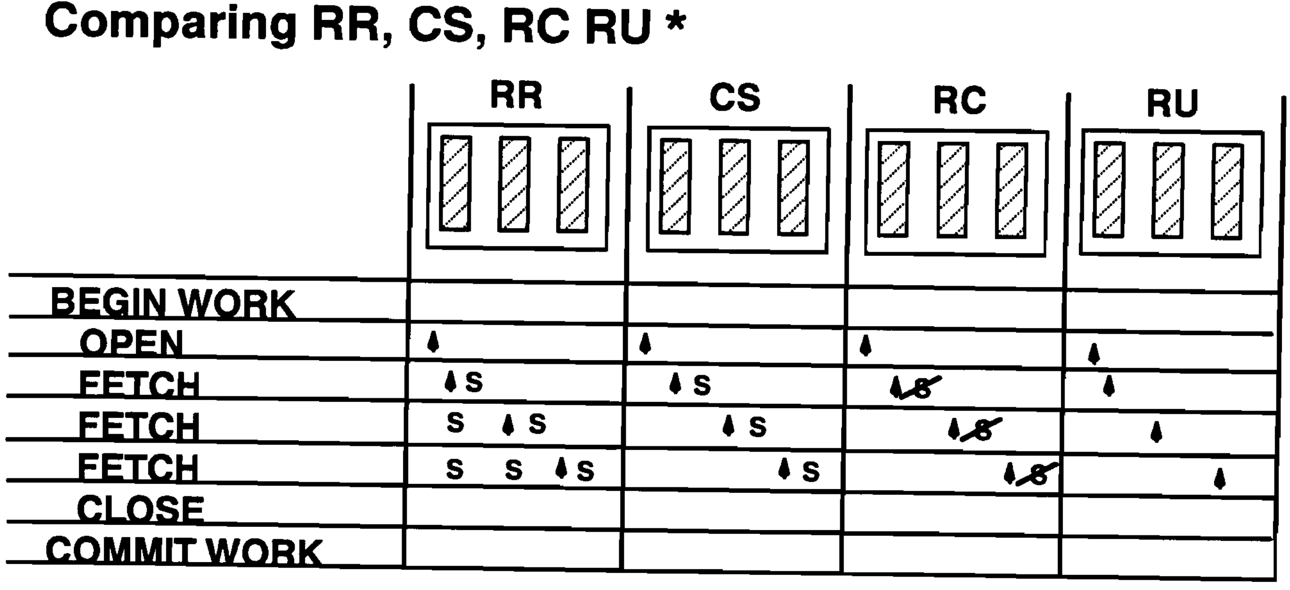


Figure 4-46.

Key Points (continued)

- If the transaction you are using has a **non-updatable cursor**, consider the following cases and use the appropriate isolation level.
  - Use the RR isolation level if the transaction needs to retain all locks until the end of the transaction. Use RR if the transaction needs to view a consistent snapshot of the data at a single point in time, especially if you need to make data modifications based on the values returned by the cursor.
  - Use the RC isolation level if the transaction simply needs to view a simple snapshot of committed data and does not need to make data modifications based on the values returned by the cursor.
  - Use the RU isolation level if it is acceptable for the transaction to scan through rows of uncommitted data in addition to committed data, and the transaction does not need to make data modifications based on the values returned by the cursor.
- If the transaction will only access data that cannot be modified by any user (you know in advance that no user will ever change the data), you can use the RU isolation level for optimal performance.

Purpose of Slide

To describe locking on the system catalog tables.

Example of Locking of System Catalog Tables

```
isql => BEGIN WORK RU;

isql => SELECT * FROM SYSTEM.TABLE;
```

NOTE:

SYSTEM.TABLE is actually a view that is built on an internal base table named HPRDBSS.TABLE, which contains a record of each table and view in the DBEnvironment.

Figure 4-47.

Key Points

- The system catalog is a set of base tables owned by the special user HPRDBSS. Views owned by the special users SYSTEM and CATALOG are defined on these tables to enable the DBA and other users to access information in the system catalog tables.
- The descriptions of isolation levels on the previous pages refer to rows in user tables, not system catalog tables. When ALLBASE/SQL reads rows in a system catalog table as part of normal internal processing, the RR isolation level is always used; otherwise, the isolation level specified in the transaction is used. When rows in a system catalog table are explicitly read by a user transaction, the isolation level specified in the transaction is used.
- For example, when queries are translated, ALLBASE/SQL needs to read rows in system catalog tables as part of normal processing:
  - In a preprocessed application program, queries are normally translated during the preprocessing phase. The locks obtained during translation are released when preprocessing completes.
  - In a dynamic application program (such as ISQL), queries are dynamically translated when the program is run. Both the locks that are required to translate a query and the the locks that are required to execute the query are obtained in the same transaction.
- Consider the example on the previous page:
  - When the `SELECT` statement is translated in ISQL, normal internal processing includes determining whether or not a table or view named SYSTEM.TABLE actually exists. ALLBASE/SQL determines this by performing an index scan (using an internally defined index) over HPRDBSS.TABLE, to try to retrieve a row for an object named SYSTEM.TABLE. If a row exists, the object exists; otherwise an error is returned.
  - During the index scan, an IS lock is obtained at the table level and an S lock is obtained on the page that contains the row (all of the system catalog tables are of type PUBLIC). These locks are retained until the end of the transaction, because the RR isolation level is used.
  - After the query has been translated, it is immediately executed. A serial scan is performed over HPRDBSS.TABLE to satisfy the needs of the query. This serial scan is not normal internal processing, it is simply a step in the access plan for this particular `SELECT`, which happens to involve a system catalog table. Therefore, the isolation level specified in the `BEGIN WORK` is used (in this case, RU), so no additional locks are acquired on HPRDBSS.TABLE.
  - If the same query exists in a preprocessed application program, the IS and S locks would be obtained and released during the preprocessing phase, and the translated version of the query would be stored as a section in the DBEnvironment. When the program is run, no locks would be obtained on HPRDBSS.TABLE when the stored section is executed.

Purpose of Slide

To describe cursors with the KEEP CURSOR option.

Understanding a Cursor Without KEEP CURSOR

Command Sequence	Xact Status	Cursor Status
OPEN; COMMIT WORK;	ENDS	CLOSED
OPEN; ROLLBACK WORK;	ENDS	CLOSED

Figure 4-48.

Key Points

- As was described previously, isolation levels can be used to release S locks prior to the end of the transaction. But by default, X locks are retained until the end of the transaction. If you want to release X locks before the transaction terminates use the KEEP CURSOR clause.
- The `OPEN` statement is used to open appropriate scans to access rows in the active set, and to position the cursor before the first row that should be fetched. The `OPEN` statement has an optional KEEP CURSOR clause that affects the cursor's behavior when a `COMMIT WORK` or a `ROLLBACK WORK` statement is issued.
- If you open a cursor without using the KEEP CURSOR clause, and you issue a `COMMIT WORK` or a `ROLLBACK WORK` statement without explicitly closing the cursor (you have not issued a `CLOSE` statement), ALLBASE/SQL will automatically close the cursor when the transaction ends. ALLBASE/SQL will also automatically close the cursor when a `ROLLBACK WORK TO SAVEPOINT` statement is issued.
  - Issuing explicit `CLOSE` statements is good programming practice, even though ALLBASE/SQL does not explicitly require them under these circumstances.

Purpose of Slide

To describe cursors opened using the KEEP CURSOR option.

Understanding a Cursor Using KEEP CURSOR

Command Sequence	Xact Status	Cursor Status
OPEN KEEP CURSOR; COMMIT WORK;	ENDS; (NEW XACT)	POSITION MAINTAINED
OPEN KEEP CURSOR; ROLLBACK WORK;	ENDS	CLOSED
OPEN KEEP CURSOR; COMMIT WORK; ROLLBACK WORK;	ENDS; (NEW XACT)	POSITION RESET
OPEN KEEP CURSOR; COMMIT WORK; CLOSE; COMMIT WORK;	ENDS	CLOSED

Figure 4-49.

Key Points

- If you open a cursor using the KEEP CURSOR clause and you issue a COMMIT WORK or a ROLLBACK WORK statement without explicitly closing the cursor, ALLBASE/SQL does not normally close the cursor:
  - If you issue a COMMIT WORK statement while the cursor is open, ALLBASE/SQL does not close the cursor. Instead, a new transaction is automatically started (a BEGIN WORK statement is implicitly issued). From the user's standpoint, it appears that the old transaction has not terminated: the new transaction has the same isolation level and priority as the transaction that was just terminated, and the cursor's position in the active set is maintained in the new transaction. For example, if a cursor's active set covers four rows, and a COMMIT WORK is issued after the first and second row had been fetched and modified, the cursor would still be pointing to the second row at the beginning of the new transaction. When the next FETCH is issued (in the new transaction), the cursor will move to the third row.
  - If you issue a ROLLBACK WORK statement while the cursor is open, one of the following occurs:
    - If you have not issued a COMMIT WORK since you issued the OPEN statement, the ROLLBACK WORK statement closes the cursor and undoes any changes made through it. The KEEP CURSOR option only takes effect if the the OPEN statement is committed.
    - If you have issued a COMMIT WORK since you issued the OPEN statement, the ROLLBACK WORK statement only rolls back the last transaction that was implicitly started, and then starts another transaction. The new transaction has the same isolation level and priority as the terminated transaction, and the cursor's position is reset to the same position that it had at the beginning of the terminated transaction (this position also is the position held at the end of the last committed transaction).
  - If you attempt to issue a ROLLBACK WORK TO SAVEPOINT statement while the cursor is open, an error will be returned. ROLLBACK WORK TO SAVEPOINT is not allowed with cursors opened with KEEP CURSOR.
- When KEEP CURSOR is used, the logical end of the transaction occurs during the COMMIT WORK statement that immediately follows the CLOSE cursor statement.
  - An explicit CLOSE statement is required if you have committed an OPEN statement that includes the KEEP CURSOR clause. If KEEP CURSOR is in effect, the cursor remains open when you issue a COMMIT WORK or a ROLLBACK WORK statement. The cursor will only be closed when a CLOSE statement is explicitly issued and a COMMIT WORK is issued.

  In ALLBASE/SQL, X locks are still retained until the end of any transaction (until a COMMIT WORK or ROLLBACK WORK is issued). The KEEP CURSOR feature actually generates many small ALLBASE/SQL transactions to create the illusion of a single, logically continuous user transaction in which X locks can be released before the end of the transaction.

Purpose of Slide

To describe cursors opened with the KEEP CURSOR option (continued).

Understanding KEEP CURSOR with NOLOCKS *

Transaction	Statement	RR	CS	RC	RU
XACT 1	BEGIN WORK				
	OPEN KEEP CURSOR	↑	↑	↑	↑
	FETCH	↑ SIX	↑ SIX	↑ ~~S~~	↑
	UPDATE WHERE CURRENT	↑ X	↑ X	↑ (X*)	↑ (X*)
	COMMIT WORK	↑	↑	↑	↑
XACT 2	FETCH	↑ SIX	↑ SIX	↑ ~~S~~	↑
	UPDATE WHERE CURRENT	↑ X	↑ X	↑ (X*)	↑ (X*)
	ROLLBACK WORK	↑	↑	↑	↑
XACT 3	FETCH	↑ SIX	↑ SIX	↑ ~~S~~	↑
	COMMIT WORK	↑	↑	↑	↑
XACT 4	FETCH	↑ SIX	↑ SIX	↑ ~~S~~	↑
	UPDATE WHERE CURRENT	↑ X	↑ X	↑ (X*)	↑ (X*)
	CLOSE CURSOR	X	X	X	X
	COMMIT WORK				

* Assuming Index Scan and UPDATABLE cursor

↑ – Cursor

SIX – SIX row lock

X – X row lock

~~S~~ – S row lock obtained and released

(X*) – Dangerous UPDATE – need REFETCH first

Figure 4-51.

Key Points

- Understanding KEEP CURSOR WITH NOLOCKS
  - When the WITH NOLOCKS clause is used, all locks (including those associated with the position of the kept cursor) are released when you issue a `COMMIT WORK` or a `ROLLBACK WORK` statement.
  - Because locks associated with the position of the kept cursor are not retained, it is possible that tables or indexes on which the cursor depends might be modified in a way that is catastrophic to the cursor. For example, a table might be dropped. In general, it is wise to disable data definition using the SQLUTIL `ALTDBE` command before using the KEEP CURSOR WITH NOLOCKS option.
  - If you open a cursor using the KEEP CURSOR WITH NOLOCKS option in an RR, CS, RC or RU transaction, and the cursor is updatable, data integrity is guaranteed (rows retrieved via a `FETCH` or a `REFETCH` statement are guaranteed to be current). It is not possible for one user to accidentally overwrite changes made by another user, as long as each transaction reviews rows that are fetched (or refetched) prior to making updates.
    - When you issue a `COMMIT WORK` statement, all locks are released. This permits other transactions to delete or modify rows in the user table. When the first `FETCH` statement following a `COMMIT WORK` statement is issued, ALLBASE/SQL will reacquire appropriate S or SIX locks (and intention locks). If RC or RU is used, these locks are immediately released, so the `REFETCH` statement is needed to acquire SIX locks prior to making an update. It is possible that another transaction could have modified the data between the time of the `COMMIT WORK` and the time of the `FETCH` (or `REFETCH`). As long as your transaction reviews rows that are fetched (or refetched) prior to making updates, data integrity will not be affected.
  - If you open a cursor using the KEEP CURSOR WITH NOLOCKS option in an RR transaction and the cursor is *not* updatable, ***data integrity is no longer guaranteed.***
    - Remember that if a transaction includes a cursor that is not based on an updatable query, a temporary table might be used for the query result. When you issue each `FETCH` statement on such a cursor, ALLBASE/SQL simply retrieves the next row from the existing temporary table.
    - If your transaction does not retain locks on the original user table, other transactions can modify it (or even drop it). Therefore, your transaction may fetch a row from the temporary table that logically no longer exists in the original table, or it may see an older version of a row that has since been modified.
    - Normally, if the RR isolation level is used for such a cursor, data integrity is guaranteed because locks are retained on the user tables from which the temporary table was created. However, if the cursor is opened using the KEEP CURSOR WITH NOLOCKS option, all locks are released when you issue each `COMMIT WORK` statement, and other transactions can modify rows in the original table. In such cases, it is the application developer's responsibility to maintain data integrity by verifying the current value of a row before updating it or using it as the basis for updating another table.

Summary of Topics Covered

TERMS

- concurrency
- deadlock
- throttled transaction
- priority
- granularity
- table types:
  - PRIVATE
  - PUBLICREAD
  - PUBLIC
  - PUBLICROW
- lock mode types:
  - (S) Share
  - (X) Exclusive
  - (IS) Intent Share
  - (IX) Intent Exclusive
  - (SIX) Share and Intent Exclusive
- intention lock
- lock strength and compatibility
- access plan
- serial scan
- index scan
- hash scan
- TID scan
- page table page
- hash key
- cursor (updatable and non-updatable)
- isolation levels:
  - (RR) Repeatable Read
  - (CS) Cursor Stability
  - (RC) Read Committed
  - (RU) Read Uncommitted

COMMANDS

- CREATE DBEFILESET
- CREATE DBEFILE
- ADD DBEFILE TO DBEFILESET
- REMOVE DBEFILE FROM DBEFILESET
- DROP DBEFILE
- DROP DBEFILESET
- CREATE TABLE
- CREATE VIEW
- CREATE GROUP
- ADD TO GROUP
- GRANT
- REVOKE
- BEGIN WORK
- COMMIT WORK
- ROLLBACK WORK
- SET USER TIMEOUT
- SAVEPOINT
- ROLLBACK WORK TO SAVEPOINT
- ALTER TABLE
- LOCK TABLE
- GENPLAN
- DECLARE CURSOR
- OPEN
    - OPEN KEEP CURSOR WITH LOCKS
    - OPEN KEEP CURSOR WITH NOLOCKS
- FETCH
- REFETCH
- UPDATE WHERE CURRENT
- DELETE WHERE CURRENT
- CLOSE