A View to a Kill

by Craig Mullins

Discussion Points

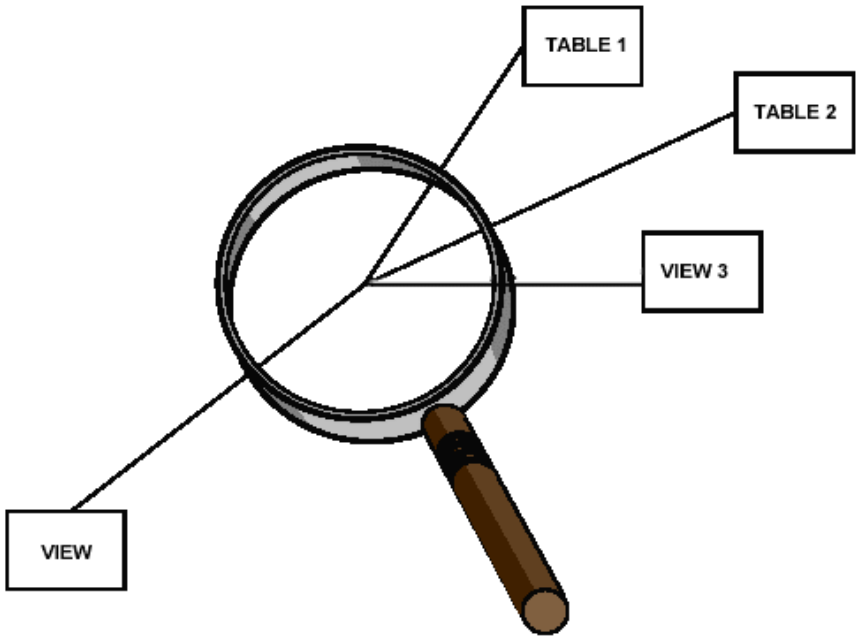
- Is it practical and/or advisable to implement base table views to insulate application programs from database change?
- What are the cost and benefits that are associated with the usage of base table views?
- Is there a relevant difference between accessing base table views and accessing base tables?
- How can I implement views in a practical, workable manner to avoid excessive administrative costs and overhead?

Contents

- **What is a view?**

A view is an alternative representation of data from one or more tables (or views). A view can include all or some of the columns contained in tables (or views) on which it is defined.

(DB2 SQL Reference)



View Overview

- **Virtual Table**

  Acts to the end user as if it were any other DB2 base table. No data is physically stored; data is accumulated from other tables only when the view is accessed.

- **Defined by SQL**

  A view is defined by the same SQL SELECT statements used to access any table. When the view is accessed, the defining SQL statements are executed.

- **Examples**

```
CREATE VIEW HIGH_PAID_EMP
EMPNO, FIRST_NAME, INITIAL, LAST_NAME,
PHONE_NO, JOB, EDUC_LVL, SEX, SALARY)
AS SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
PHONENO, JOB, EDLEVEL, SEX, SALARY
FROM EMP
WHERE SALARY > 50000;

SELECT * FROM HIGH_PAID_EMP;
```

- Views cannot contain:
  - FOR UPDATE OF
  - UNION / UNION ALL
  - ORDER BY
  - FOR FETCH ONLY
  - OPTIMIZE FOR . . .

- Certain Views Can Not Be Updated (I/U/D):
  - Joins
  - Column Functions
  - DISTINCT

- GROUP BY / HAVING
- Subquery w/ Same Inner & Outer Table
- View Based on Read Only View

- Certain Views Can Not Be Inserted To:
  - Derived Data *
  - Constants
  - Views Defined Without All
    - Columns Not Having a Default

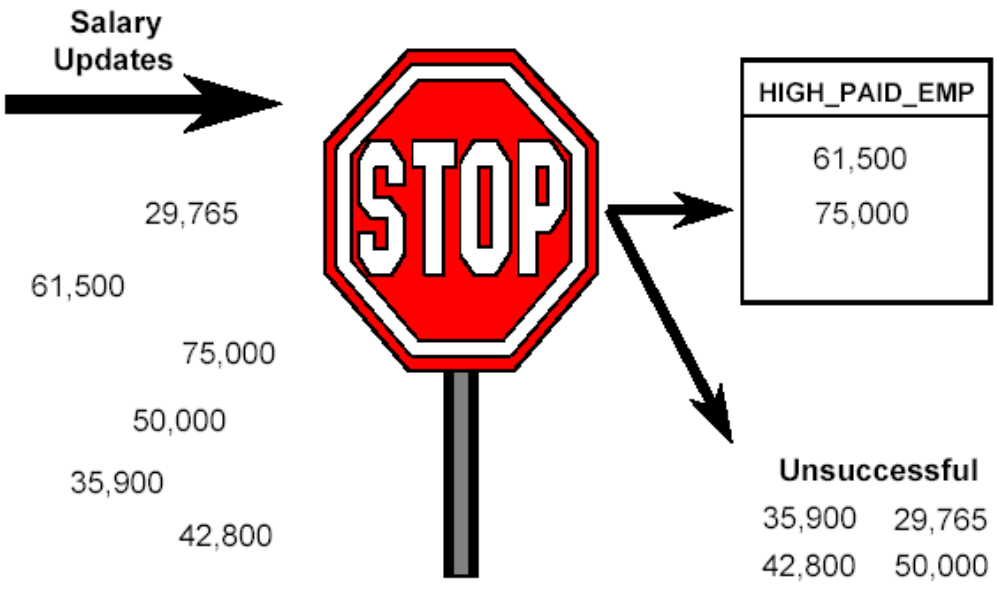
(that is either nullable or NOT NULL WITH DEFAULT)

** Arithmetic Expressions*

- **With Check Option**

Ensures view update integrity.

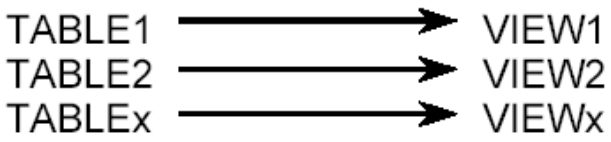
```
 CREATE VIEW HIGH_PAID_EMP
     (EMPNO, FIRST_NAME, INITIAL, LAST_NAME,
      PHONE_NO, JOB, EDUC_LVL, SEX, SALARY)
AS SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
          PHONENO, JOB, EDLEVEL, SEX, SALARY
   FROM   EMP
   WHERE  SALARY > 50000
WITH CHECK OPTION;
```



Top

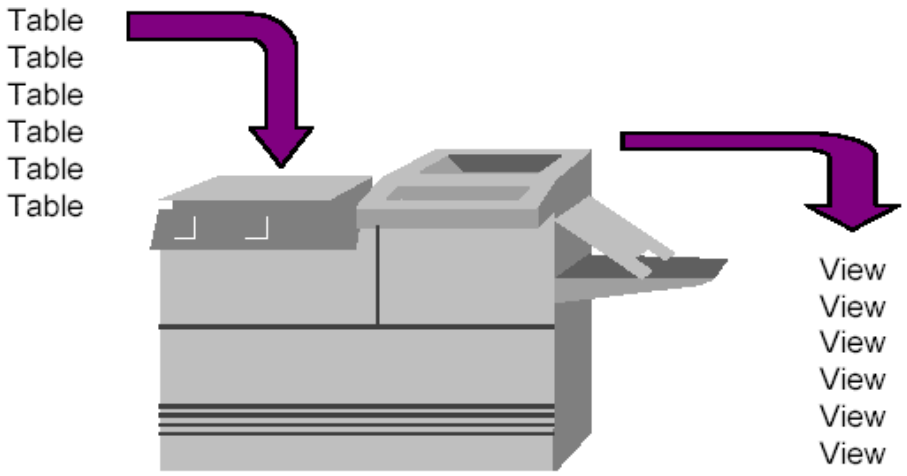
Base Table View Rationale

- **One View For Each Base Table**



- **Theory: All application programs should access views only, never base tables**

This (supposedly) insulates application programs from the effects of database change.



- **But Is It All True ?**

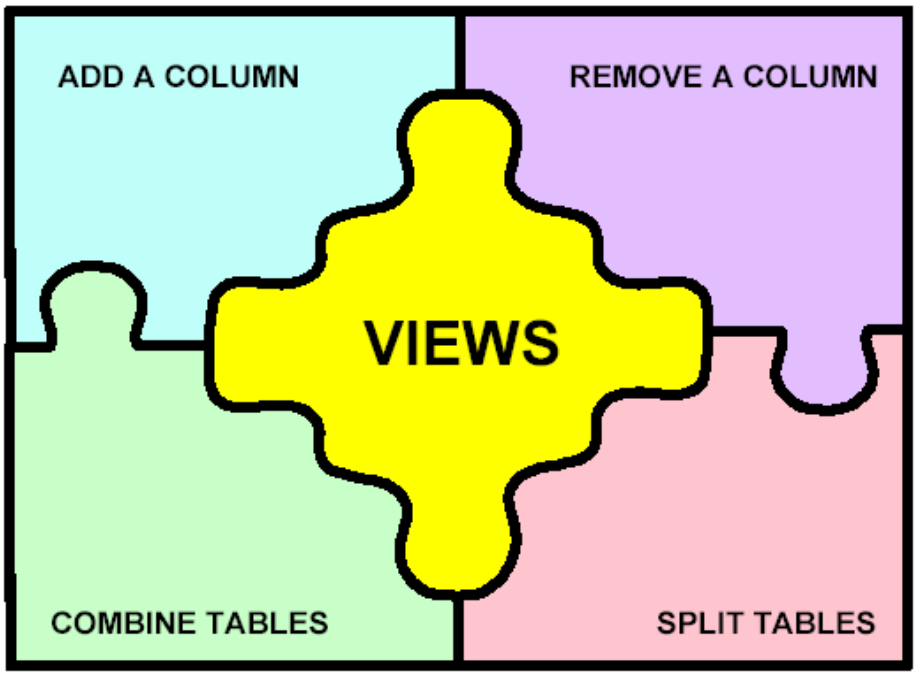
  Maybe to a very limited degree, but not really. The administrative burden is almost never worth the increased maintenance costs.

  Let's see why . . .

Top

Application Program Insulation

There are four types of database change that base table views purport to insulate application programs from:

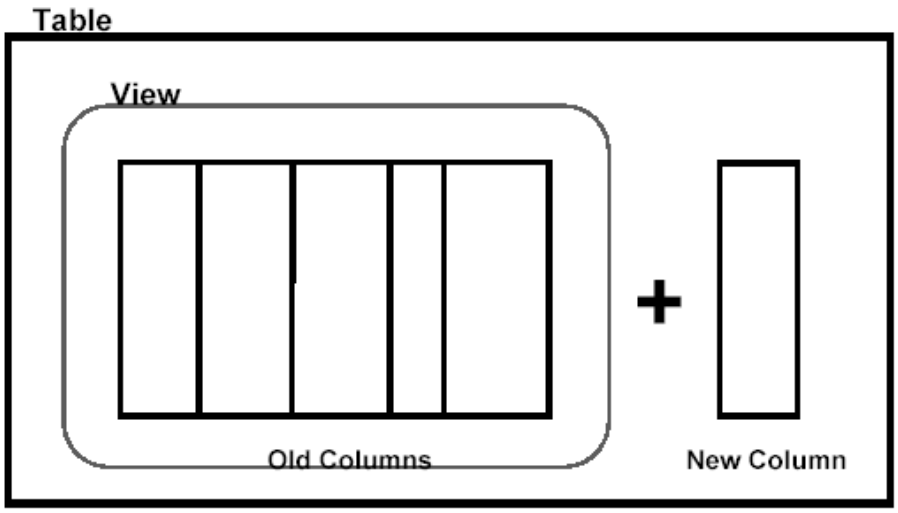


- Adding columns to a table
- Removing columns from a table
- Combining multiple tables into a single table
- Splitting one table into multiple tables

- Application Program Insulation
- Adding a column

Theory: when you need to add a column, the column can be added without affecting application programs because they all access views, not the base table.

- Must never change the view.
- Must create new views for the new column to be usable by any new (or existing) application programs.



- **Adding a column**

  Analysis of Effects

- If the column is not added to the view, all SQL SELECT statements on the base table view will continue to operate correctly.

- All INSERT statements will work, but only if the new column is defined as nullable or NOT NULL WITH DEFAULT.

- An insert to the view will cause the default value to be used for the new column that is not in the view.

- All existing UPDATE and DELETE statements will function properly.

- Adding a column

  But . . .

  If you always explicitly list each column in your application program's SELECT and INSERT statements then all of the assertions on the previous page apply to base tables as well as to views.

  And what if you decide to change the view to add the new column?

  Then the SELECT * from the view and the INSERT to the view without a column list will not work either.

  Simple rules:

  - Never use SELECT *
  - Always explicitly name columns

- **Removing a column**

  If the column is replaced by a constant (or a calculation) in the view, then application programs need not be changed.

  But . . .

  Unable to INSERT into a view with constants.

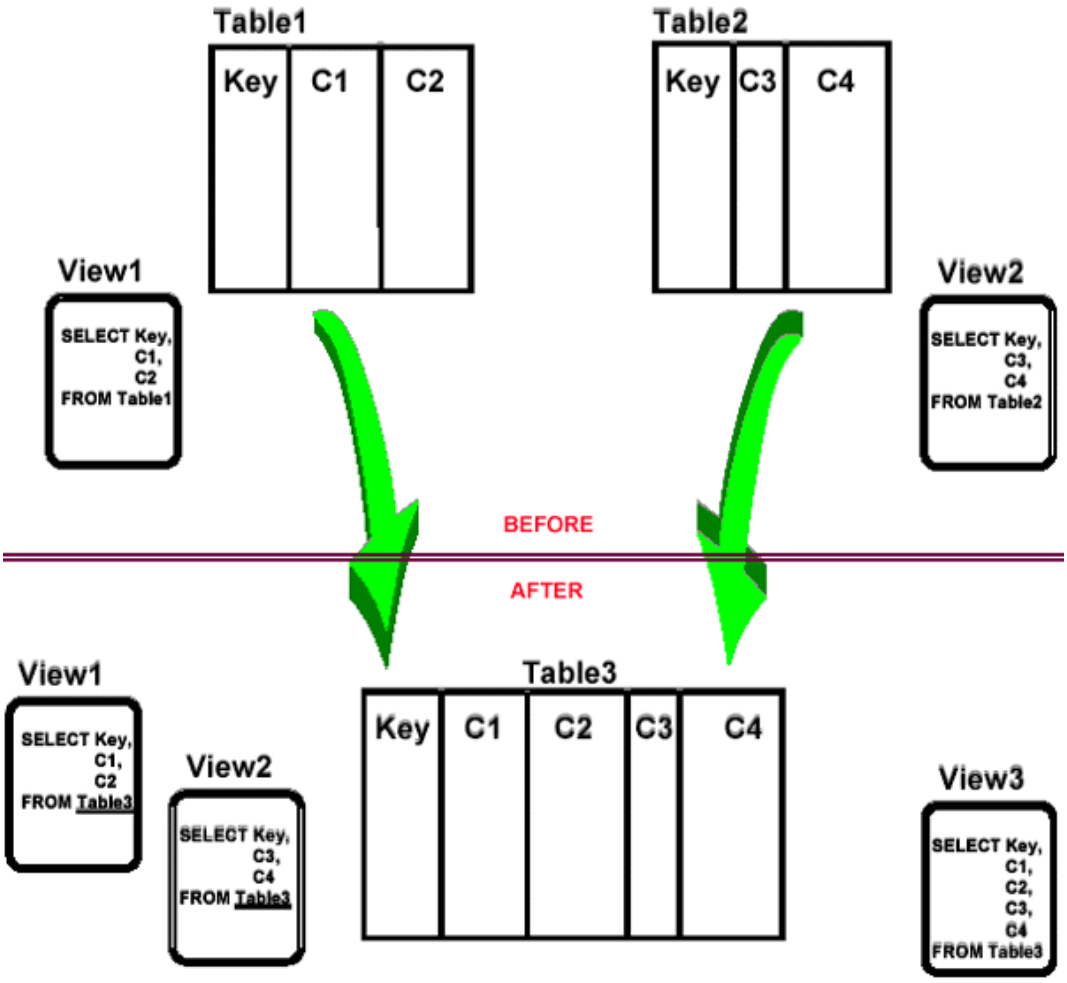
  All retrievals must be reviewed! Can not rely upon a constant in production. What about:

  - report totals
  - any report detail relying upon the constant
  - INS/UPD/DEL based upon constant
  - calculations using the constant

    Final Verdict: **Accuracy May Suffer!**Removing a column

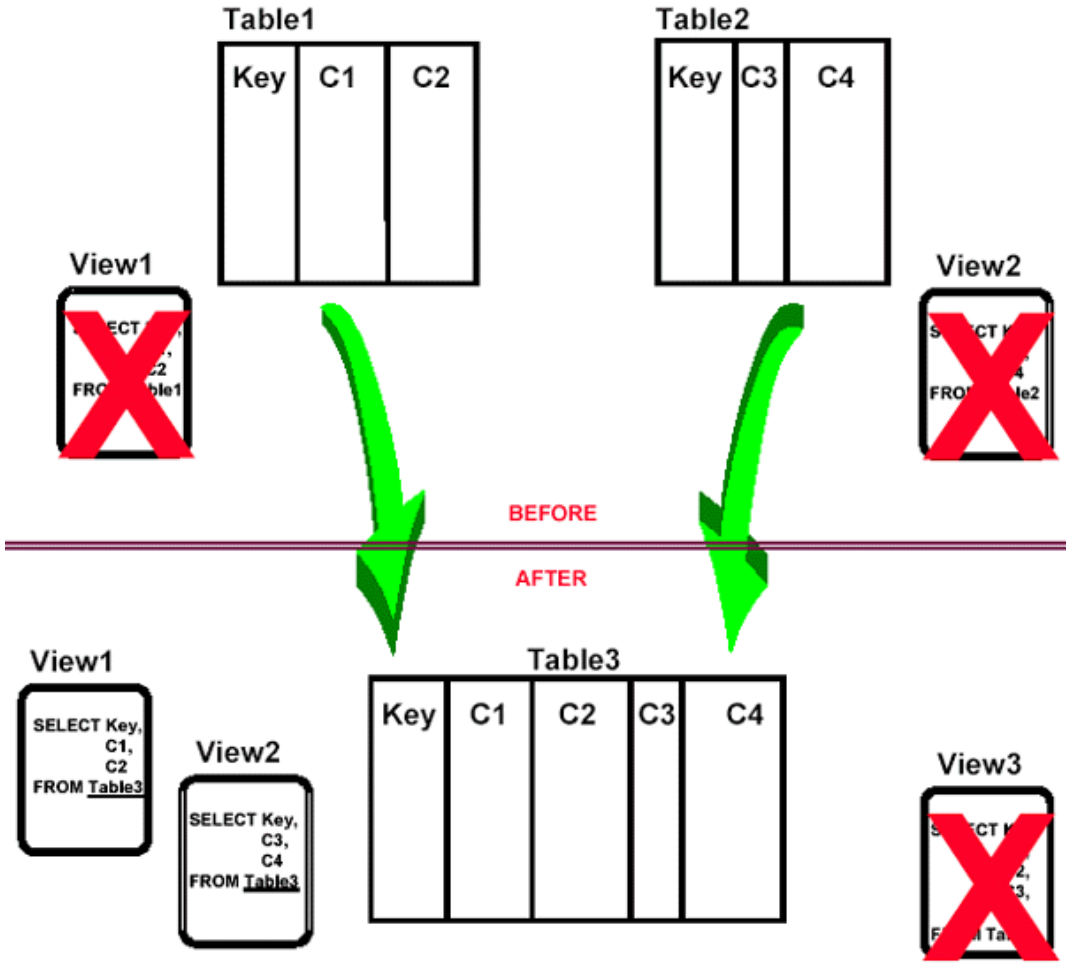
- The bottom line is whenever a column is removed from a table:
  - the column must also be removed from any base table views
  - all application programs using that column must be modified
  - base table views buy you absolutely nothing in this case

- Combining tables - usual proposed scenario.



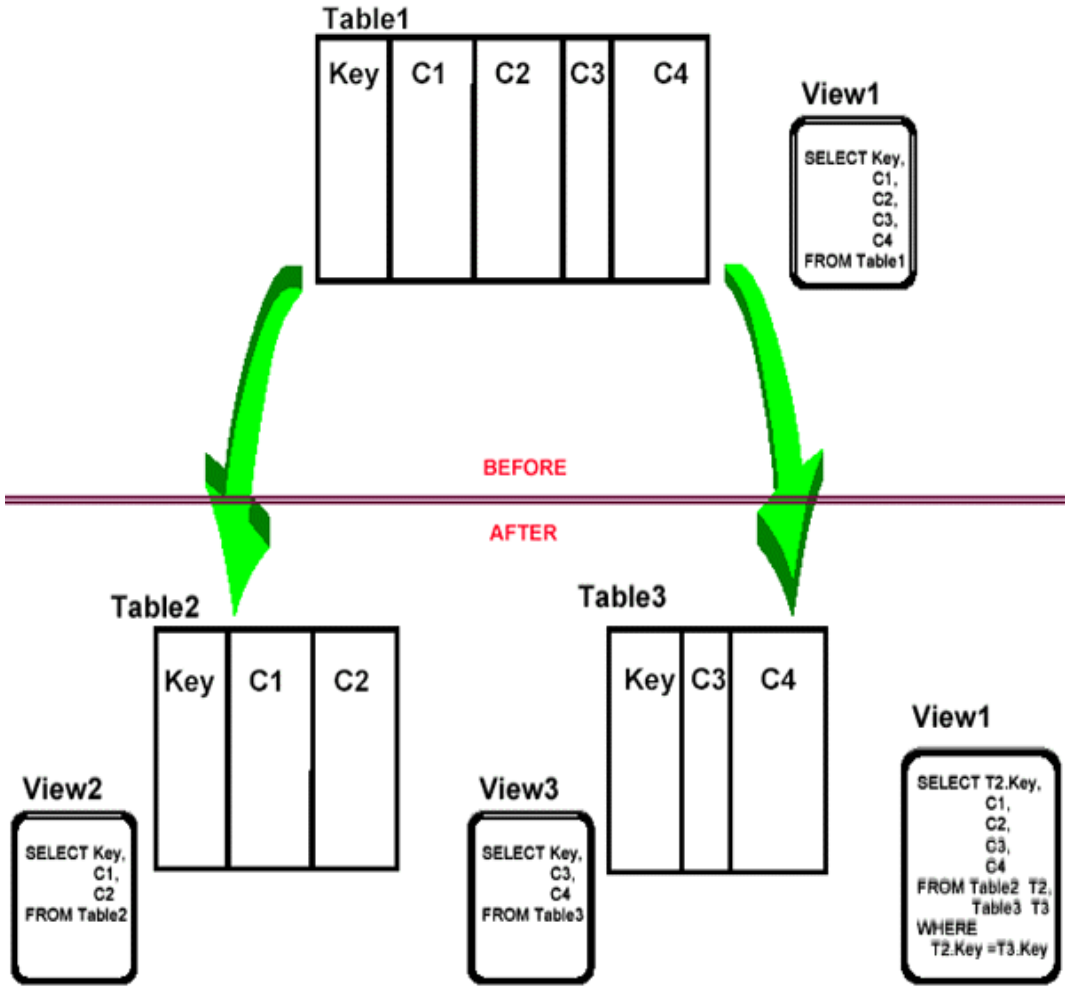
- Combining tables - the usual scenario
    - To make either of the 2 new views updateable, default values may have to be changed (ie. NNWD to nullable). Is this acceptable?
    - View1 and View2 will need to be dropped and re-created with a new FROM clause.
    - How did having View1 and View2 prior to the database change help us ?
    - Is this really useful?
- Combining tables - why not like this ?

- **Combining tables** - why not like this ?



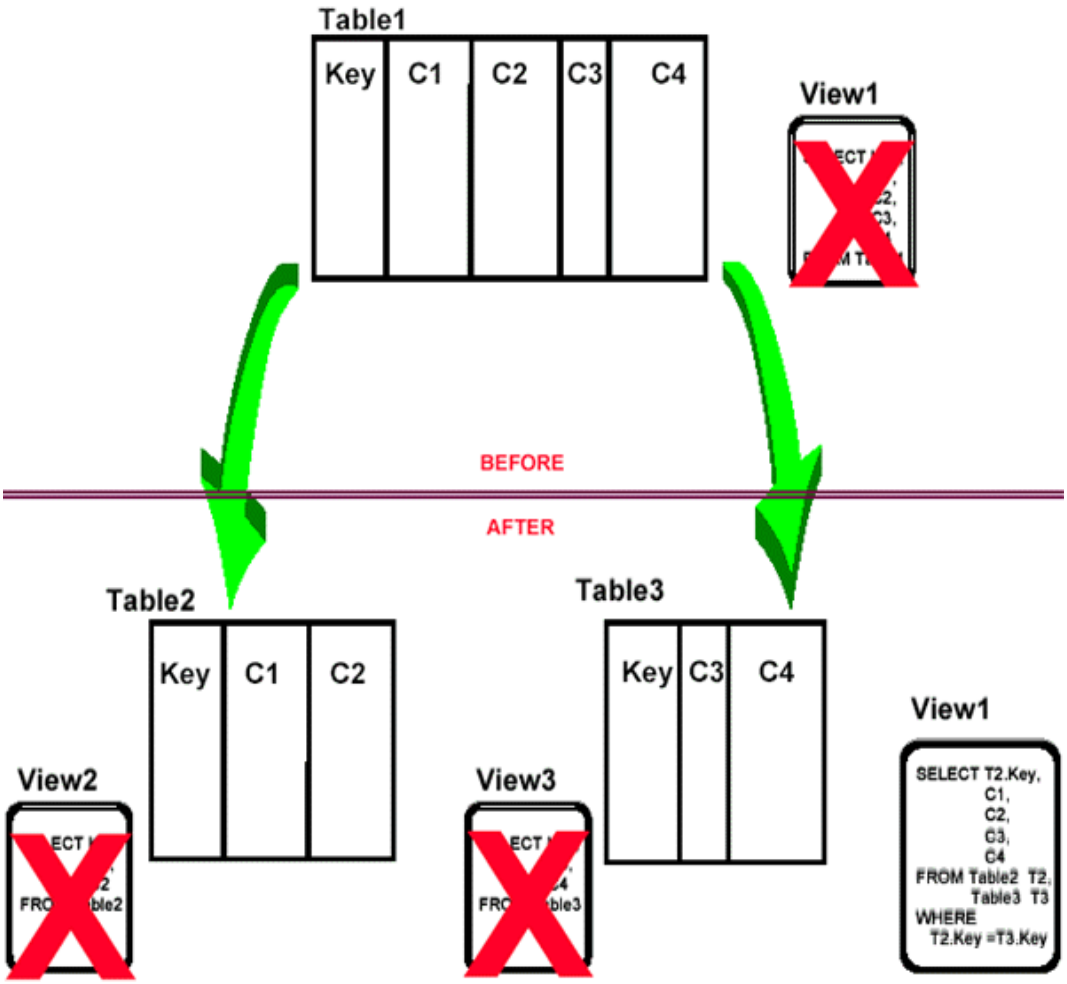
- **Combining tables**
  - avoid using views until absolutely necessary
  - avoid the costly overhead of maintaining base table views
  - new views can be named the same as the old tables that they are replacing; avoids recoding
  - let's face it, poor performance is just about the only compelling reason to combine tables; perhaps better pre-production testing methods or a faster machine can solve this problem, too
    Does the computer rule your business?

- **Splitting tables** - the usual scenario



- **Splitting tables** - the usual scenario
  - The revised view, VIEW1, is not updateable.
  - What about old statements that access only (Key, C1, C2) or (Key, C3, C4) ?

    Without program changes, these requests to access VIEW1 will cause an unnecessary join to be done adversely impacting performance!
  - How did having VIEW1 prior to the database change help us ?
  - Is this really useful ?
- **Splitting tables** - why not like this ?



- **Splitting tables**
  - avoid using views until absolutely necessary
  - avoid the costly overhead of maintaining base table views
  - new view can be named the same as the old table that it is replacing; avoids recoding
  - without programming changes performance will suffer due to the new join
  - let's face it, this does not happen very often; if it does happen often, you need better data administration and testing efforts

  Demand better DA and testing efforts ! ! !

- Synopsis
  - Adding a Column

    When you add a column to a table, presumably someone needs to use that column. The old base table view can not, unless it is changed. Therefore, you must either change the view or create an additional, new base table view.

  - Removing a Column

When a column is removed from a base table it must also be removed from all base table views or replaced by a constant. But using constants can adversely impact system and data integrity.

- Combining and Splitting Tables

  A change of this magnitude requires a thorough analysis of your application code. Views can be created after the split or combination to reduce the impact of the change. They are not needed prior to that !

Top

Q: What is the best strategy for maintaining your organization's base table views?

a. Modify the view every time that the underlying base table changes.

b. Create a new base table view every time that the underlying base table changes.

c. Avoid base table views.

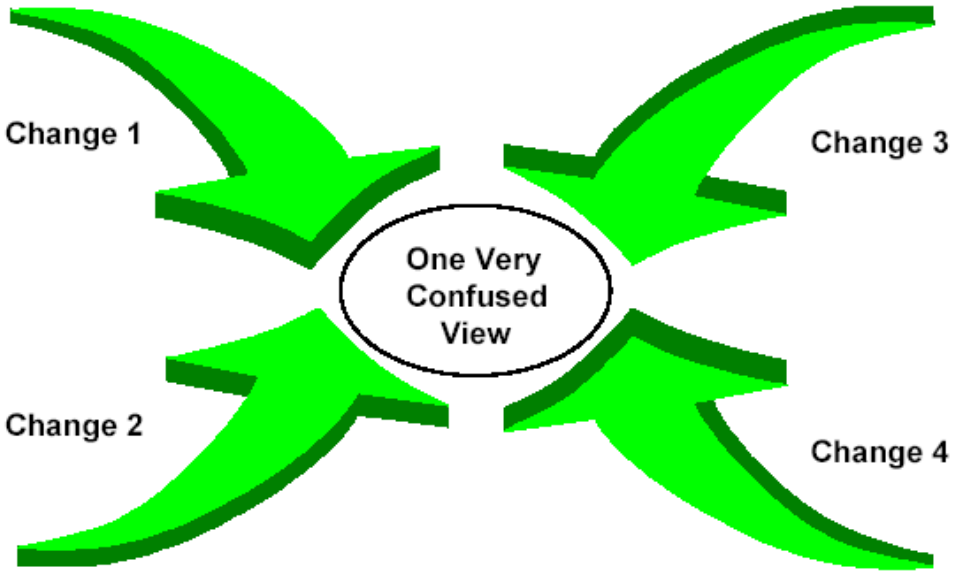
d. Make the new guy worry about it, I've got real work to do !

e. Sometimes a, sometimes b.

f. None of the above.

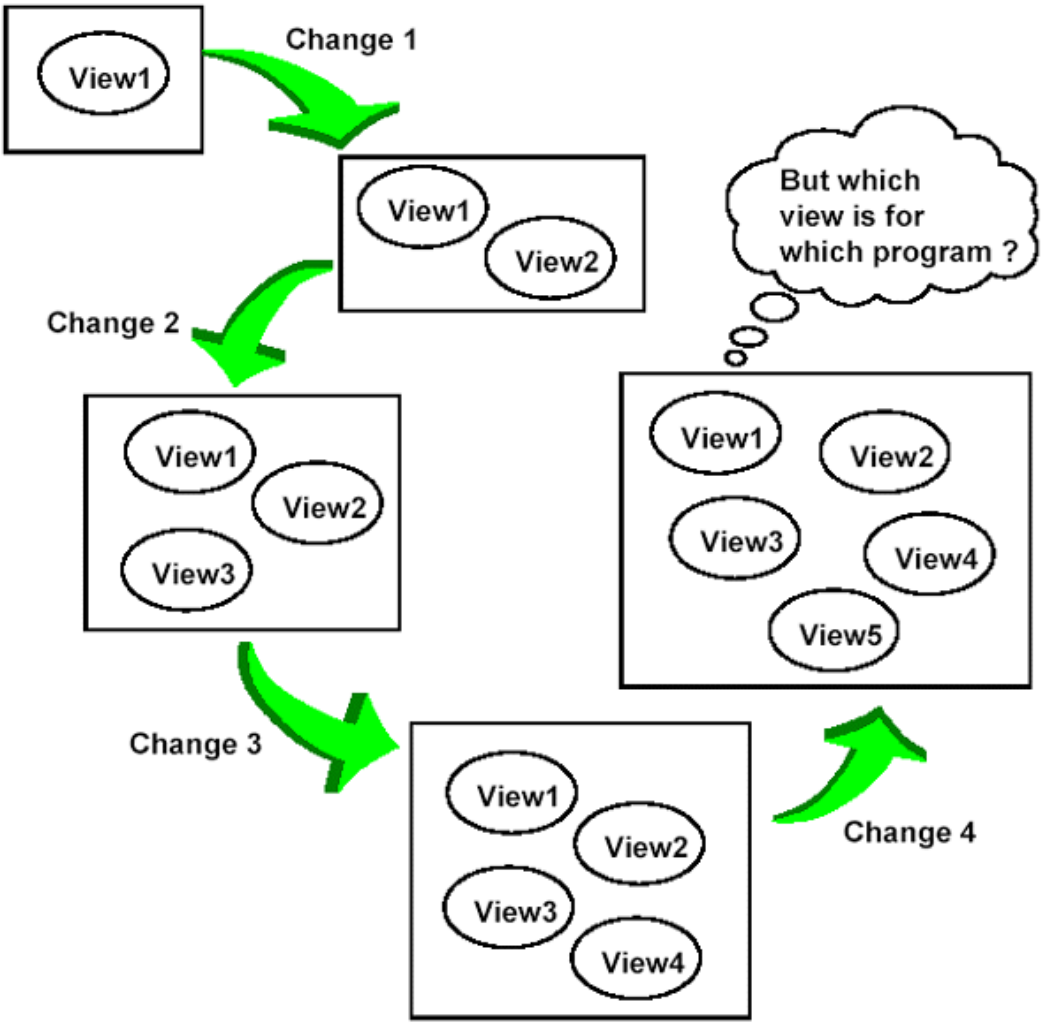
A: d *Not really, I say it is* c

Scenario 1: Modify the base table view for each change



- For each change, drop and create the view to reflect the change.
- But does this provide insulation? How is it different than changing the table?

Scenario 2: Create a new view for each change



Scenario 2: Create a new view for each change

- How long do we continue to create these new views? Do we ever clean up by dropping old views that are no longer needed or used?
- Who adminsters changes to these views? If change 3 drops a column, then views 1, 2, 3, and 4 must all reflect this change!
- Who will know which view to use for which application program? Ad hoc? Always the newest? Always the original view used by the program? A combination?
- You must maintain an explosion of views!
- WHAT A MESS ! ! !

Scenario 3: Avoid base table views

- Do not use one view per base table and force your application programmers to use them when coding.
- Practice data modeling techniques to reduce the number of database changes.
- Perform adequate application testing before moving to a production

environment.

- Unit testing
- Integration testing
- System testing
- Quality Assurance testing
- User Acceptance testing
- Stress testing

- Use views only after major database changes occur (such as splitting and combining tables).

  Note: These are NOT base table views, but views that make current views look like old base tables that are no longer on the system!

Scenario 3: Avoid base table views

- Code programs to avoid SELECT * and to explicitly name all columns needed in every embedded SELECT and INSERT statement.
-

use
```
SELECT C1, C2, C3, C4
FROM TABLE1
```

not
```
SELECT * FROM TABLE1
```

use
```
INSERT (C1, C2, C3)
VALUES (V1, V2, V3)
INTO TABLE1
```

not
```
INSERT VALUES (V1,
V2, V3)
INTO TABLE1
```

- Avoiding SELECT * should not be a big problem because most shops already impose this as a standard whether or not they use base table views... I wonder why ?

Scenario 3: Avoid base table views

- Do not use arbitrary naming standards that force you to name views differently than tables. For example, do not embed a "V" into your view names: use the same naming convention for all table-like objects.
  - a view is a logical table
  - SQL (syntactically) operates on views the same as it operates on base tables.
  - for technical people to differentiate between tables and views, they can query the DB2 Catalog
  - User-friendly view names are especially important for dynamic SQL users (QMF, 4GL, etc)
  - Also, apply the same table naming conventions not only to views, but to aliases and synonyms, as well !

Top

View Implementation Rules

- There are three basic rules that should be followed before deciding to implement any view:
  1. View Usage Rule
  2. Proliferation Avoidance Rule
  3. View Synchronization Rule

- Following these three rules will result in an effective view implementation and usage strategy for your organization.

- It will also ensure that the generic rule of one view per base table is avoided.

- **View Usage Rule**

  Create a view only when it achieves a specific, rational goal. Each view must have a specific application or business requirement that it fulfills before it is created.

- There are seven basic uses for which views excel. These are:
  1. to provide row and column level security
  2. to ensure efficient access paths
  3. to mask complexity from the user
  4. to ensure proper data derivation
  5. to provide domain support
  6. to rename columns, and
  7. to provide solutions which can not be accomplished without views

  Orson Welles Approach to View Creation: We will create no view before its time!

- **View Usage Rule**

  Examples of each usage rule follow:

  1. provide proper predicates to provide row level security; remove columns from the view SELECT-list to provide column level security
  2. code join criteria, stage 1 and indexable predicates, etc. to ensure efficient access paths
  3. code complex joins and subselects into the view to mask complexity from the user
  4. code formulae (ie. BALANCE*INT_RATE) to ensure proper data derivation
  5. code proper predicates and use the WITH CHECK OPTION to provide domain support
  6. if desired, simply rename columns in the view SELECT-list
  7. an example of a solution which can not be accomplished without views is shown on the following foil

- **View Usage Rule**

  An example of a solution which can not be accomplished without views: showing detail and summary information on the same report! For example, create a report showing all column in a table along with aggregate information on column the columns in that table.

To accomplish:

```
CREATE VIEW COL_LENGTH
  (TABLE_NAME, MAX_LENGTH,
   MIN_LENGTH, AVG_LENGTH)
AS SELECT TBNAME, MAX(LENGTH),
          MIN(LENGTH), AVG(LENGTH)
```

```
FROM      SYSIBM.SYSCOLUMNS
GROUP BY TBNAME
```

After creating the view, the following SELECT statement provides both detail & aggregate column information:

```
SELECT TBNAME, NAME, COLNO, LENGTH,
       MAX_LENGTH, MIN_LENGTH,
       AVG_LENGTH,
       LENGTH - AVG_COL_LENGTH
FROM   SYSIBM.SYSCOLUMNS C,
       authid.COL_LENGTH V
WHERE  C.TBNAME = V.TABLE_NAME
ORDER BY 1, 3
```

- **Proliferation Avoidance Rule**

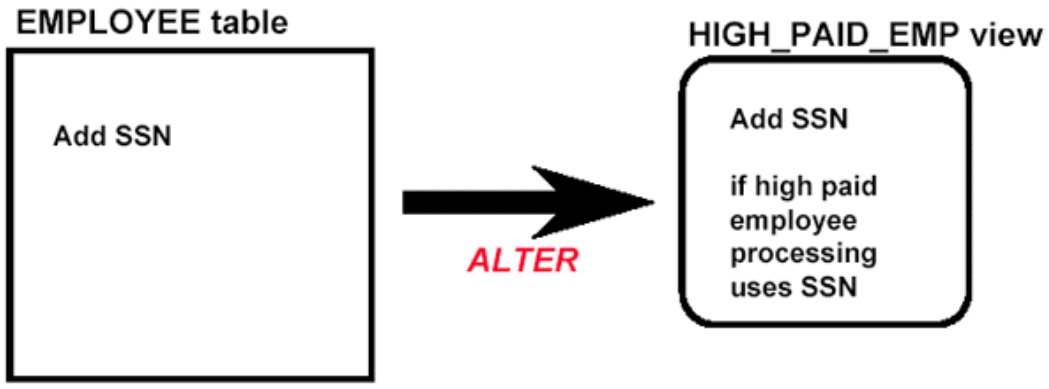
  Do not needlessly proliferate DB2 objects, in this case, views. Every DB2 object that is created constitutes additional entries in the DB2 Catalog.

  Why create what is not necessary?

- **The View Synchronization Rule Keep all of the views within your system logically pure by synchronizing them with the underlying base tables.**

  As changes occur to base tables, views may need to change as well. Make these changes based upon usage criteria to ensure that your organization has a sound, usable physical database implementation.

  Not necessary to correct each view right away. This is where views provide great logical data independence !



Top

Examples of "Good" Views

- **Security**

```
CREATE VIEW MY_PROJECTS
AS   SELECT   PROJNO, PROJNAME, DEPTNO,
              PRSTAFF, PRSTDATE, PRENDATE,
              MAJPROJ
     FROM     DSN8230.PROJ
     WHERE    RESPEMP = USER
```

This view allows the current user to view only his/her projects.

- **Access**

```
CREATE VIEW EMP_DEPTS
```

```
AS       SELECT       EMPNO, FIRSTNME, MIDINIT,
                      LASTNAME, DEPTNO, DEPTNAME
         FROM         DSN8230.EMP,
                      DSN8230.DEPT
         WHERE        DEPTNO = WORKDEPT
```

This view joins two tables efficiently.

- **Derived Data**

```
CREATE VIEW EMP_TOT_COMPENS
      (EMPNO, TOTAL_COMPENS)
AS    SELECT EMPNO, SALARY+BONUS+COMM
      FROM DSN8230.EMP
```

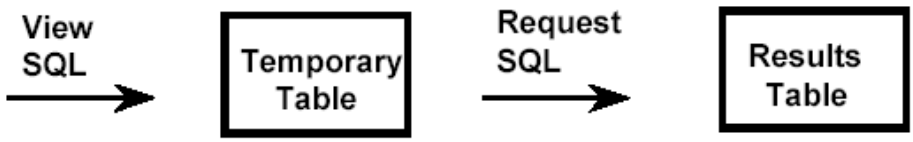
This view uses a formula to derive business data (total compensation).

Top

View Performance

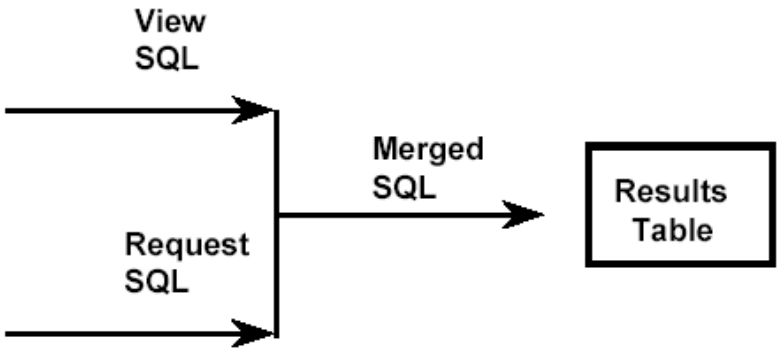
- **View Materialization**

  At run time, if view materialization is used, DB2 actually creates a temporary table containing the data for the view.



- **View Merge**

  At run time, if view merge is used, DB2 will combine the predicates from the view with the predicates from the SQL accessing the view before execution. No temporary table is required.



- View Materialization
  - Can be very costly.
  - Determine if it is being used by running EXPLAIN and checking for the view name in the TNAME column of the PLAN_TABLE.
  - Try to avoid.
  - Consult the chart to determine when view materialization will be used.

If the SELECT against the view uses a...	If the CREATE VIEW uses a...			
	GROUP BY	DISTINCT	Column Fnc	Column Fnc DISTINCT
Join	YES	YES	YES	YES
GROUP BY	YES	YES	YES	YES
DISTINCT	NO	YES	NO	YES
Column Function	YES	YES	YES	YES
Column Function DISTINCT	YES	YES	YES	YES
SELECT Subset of View Columns	NO	YES	NO	NO

All other view access will be by view merge.

- **View Merge**
  - Performs much better than view materialization.
  - If TNAME shows the underlying table, then view merge was used.
- **General Performance**
  - If view merge is used, views should perform as well as base tables.
  - Minor overhead at BIND time to perform the merging of the view.

Top

View Management Queries

- **To find all views dependent on a table that is to be changed:**

```
SELECT  DCREATOR, DNAME
FROM    SYSIBM.SYSVIEWDEP
WHERE   BCREATOR = 'Table Creator'
AND     BNAME = 'Table Name'
```

- **To find all QMF queries that access a given view:**

```
SELECT  DISTINCT OWNER, NAME, TYPE
FROM            Q.OBJECT_DATA
WHERE   APPLDATA LIKE '%viewname%'
```

- **To find all plans dependent on a table that is to be changed:**

```
SELECT  DNAME
FROM            SYSIBM.SYSPLANDEP
WHERE   BCREATOR = 'View Creator'
AND             BNAME = 'View Name'
```

- **To find all potential dynamic SQL users:**

```
SELECT  GRANTEE
FROM    SYSIBM.SYSTABAUTH
WHERE   TCREATOR = 'Table Creator'
AND             TTNAME = 'Table Name'
```

Top

Other View Arguments

- **I just feel safer using views.**

  Well, I just feel safer when I just stay in bed all day, but my wife won't let me do that ! She makes me go to work...

- **I've already implemented base table views for all of my other systems. I don't want to subvert my standards now.**

  Neither did Detroit; now look what foreign automobile producers did to them!

- **Third party tools make administering views much easier, so why not use them?**

  Aspirin makes the pain from a headache go away, so why not ?

Top

More View Arguments

- **But all the experts say to use one view per base table!**

  All the experts told Columbus that the world was flat. Too bad he didn't listen, huh?

- **I'd rather be prepared by having views before I need them.**

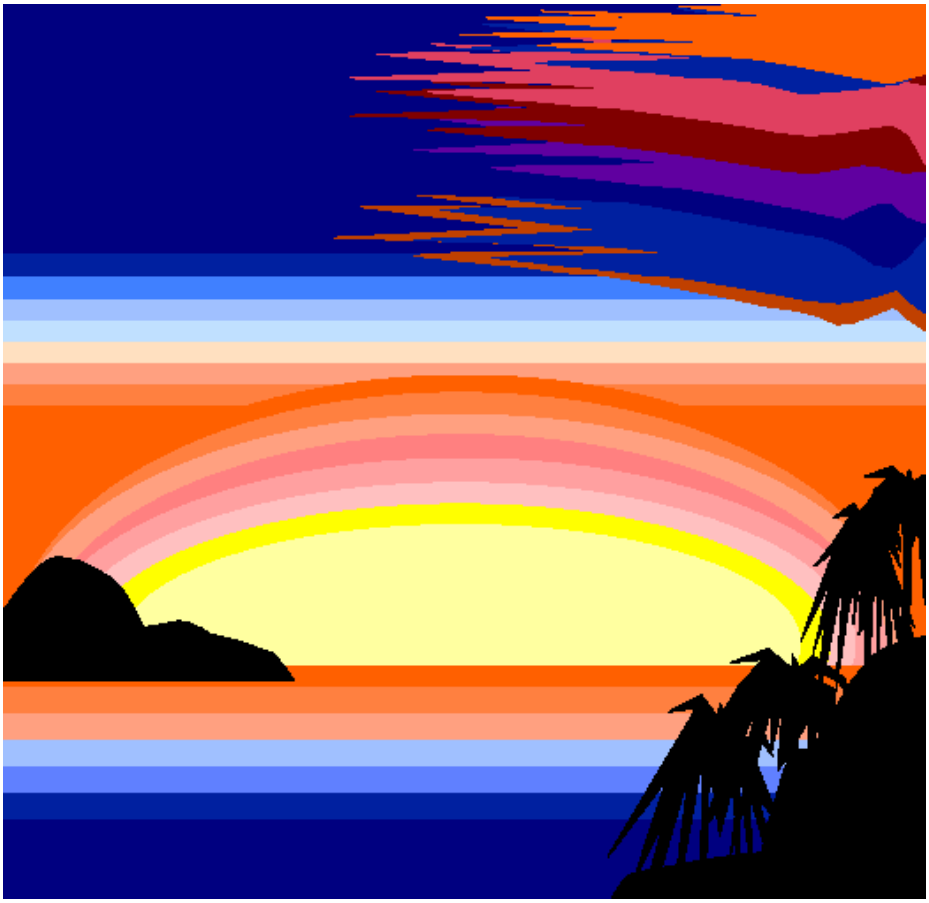
  Do you buy a car with a tow-truck attached? Does it have to have a lifetime supply of gasoline? No? Why not? If it only came with a tow truck and 2 million gallons of gasoline.

Top

References

- One View Per Base Table? Don't Do It!, by Craig S. Mullins, February 1991, Database Programming & Design,
- Views on Views, by Craig S. Mullins, DB2 Update, 1994.
- IBM DB2 for OS/390 SQL Reference
- IBM DB2 for OS/390 Administration Guide
- IBM DB2 Application Programming and SQL Guide
- DB2 Developer's Guide, by Craig S. Mullins, SAMS Publishing, for more information: (800) 428-5331
- http://www.craigsmullins.com

And in the end . . .



all that remains is a beautiful view !

--

Craig S. Mullins is director of technology planning for BMC Software. He has 20 years of experience dealing with data and database technologies. He is the author of the books *Database Administration: The Complete Guide to Practices and Procedures* and the best-selling *DB2 book, DB2 Developer's Guide* (currently in its fifth edition). Craig can be reached via his Web site at www.craigsmullins.com or at craig_mullins@bmc.com.

It's RAD - Recovery, Audit, Data Migration Plus - Using a DB2 Log
(Tuesday July 20, 1:00 pm CST)