May 15, 1996

IBM DataPropagator Relational application guide

by
Rob Goldring

International Business Machines Corporation
Software Solutions
Santa Teresa Laboratory
San Jose, California

Revisions

March 15, 1994	Added sections for generating anonymous test data, for creating large DB2 copies and for supporting logical business transactions
May 4, 1994	Added text to further explain Figure 6 relating to R.I., corrected errors in the *Equijoin of co-located source tables* section, added new sections *When adding columns to a user table* and *Automating the large copy load*.
May 19, 1994	Improved UNION definition technique to support auto-refresh by defining subset views on copies. Work-around to allow reference to before-image columns in subscription predicates. How to define joins on foreign key columns which may be updated.
May 23, 1994	Corrected work-around #2 for copy-owner naming restriction, included JCL and sample program for automating DSN1COPY refresh, added a discussion of the "re-init" Capture/MVS command relative to the timing of the ALTERing of a table to add a new column.
June 16, 1994	Corrected the view definitions in the "Equijoin on primary key columns of co-located source tables" section. Business transaction section temporarily removed pending rewriting.
June 25, 1994	Added pictorial examples of various *control server* placement schemes.
June 28, 1994	Restored "Propagating from DB2 by logical business transaction" section.
July 30, 1994	Added additional text to "Auditing" section. Added mention of DPROPR FORUM reports of experiences copying from SQL/DS using Apply/MVS and Apply/2.
August 3, 1994	Added introductory discussion to referential integrity section. Added DDL for control tables and copy tables.
September 24	Added a new section "Automated pruning of IBMSNAP_TRAIL". Improved the "DB2/2 to DB2 Copying" section. Added performance tip to use CLOSE=NO for DB2 tablespaces and indexes. Added note to "Equijoin on primary key columns of co-located source tables" regarding possible special handling required for deletes. Removed section "Copying to DB2/6000" as the capability will shortly be fully supported with the general availability of Apply/6000. Added cautionary note in section "Propagating from DB2 by logical business transaction" regarding the synchronization of change capture and application completion posting.
November 7, 1994	Fixed up "Auditing" section. Added new sections: "Work-around for the 17-byte-unique column name restriction", "Detecting DB2 utility operations", "Automating the stopping of Apply/2", "When potential registrars, subscribers not listed", "Querying recent Apply activity".
November 21, 1994	Add section "Managing magnetic and optical storage for the same DB2 table" to discuss a potential solution for a medical records application. Mention of CD reorganization tips added to the "Performance tips" section. Added note to section "Outer join of two or more source tables with common primary".
December 2, 1994	Edited "Where to place subscription control tables?" section to advise COPY SERVER placement, generally, for performance reasons.
December 7, 1994	Replaced the "Two-way propagation" section, which was in error.
April 6, 1995	Added section "Propagating to DB2/6000 Parallel Edition".
May 30, 1995	Added new sections "When pruning doesn't seem to be effective", "Protecting against an abnormally large apply volume", "Database Partitioning", "Avoiding Capture/MVS cold starts", clarified gap detection for views, hinted at coming changes in the security model and the migration of subscription control information.

June 27, 1995	Removed "not been tested" disclaimer from "Protecting against an abnormally large apply volume" technique, noted restrictive assumptions in the "Database Partitioning" example which uses subqueries instead of views, made a note in the "Automating the large copy load" section to have Capture/MVS running prior to using the procedure. Correction to the "Automating the stopping of Apply/2" procedure.
July 2, 1995	Added new sections "When your table partitioning (filtering) key is updatable", and "How to define CD-views to make updates into deletes+inserts".
August 15+16, 1995	Revised "Database Partitioning" section to use join registrations and subscriptions, thereby accommodating the cases where partitioning migration is driven solely by manipulation of the BRANCH table. This section moved under the "Advanced Tricks" section, along with the other join-view scenarios. Removed the section "Propagating from DB2 by logical business transaction".
September 5, 1995	Added Figure 4, Figure 5 and Figure 6 beginning on page 9, illustrating usage of both *internal* and *remote* CCD tables. Also updated section "Copying from DB2/2 or DB2/6000 to DB2 or DB2/400" to reflect the August 25, 1995 code refresh. Also added Figure 23 to the section on "Database Partitioning".
September 29, 1995	Some minor edits. Modified cautionary note to say that CD_OLDEST_CHANGE is not maintained for view registrations, only for customers with an edition of Capture/MVS prior to the August 25, 1995 refresh. Noted in "Target site UNIONS -- " section that CCDs should not be targets of multiple subscriptions, as is the case when maintaining unions. Added a new section: "When copying to Oracle and Sybase through DataJoiner".
October 10, 1995	Major revision, replacing all join and union examples to show use of new (still experimental) OS/2 commands JOINREG, UNISUB and UNION. These tools also added to the DPROPRAG PACKAGE, the package containing this document.
November 14, 1995	Added warning against use of DPropR/2 cancel registration for *view* registrations. Added to "Controlling the *timing* of copying" section. Added a new section "Defining views over CCD tables".
May 15, 1996	This revision includes updates for DPropR 1.2.1 and the DPRTOOLS. Update to the "Performance tips" section to include recommended IBMSNAP_UOW table indexes. Minor updates to "Registration" and "Subscription" sections. Additional text in "Data Server Placement" area of "Where to place subscription control tables?" section. Add a new, temporary section "Concerns for customers with Apply/MVS 1.2.1" Additional text in "Propagating to DB2/600 Parallel Edition" and "Copying to/from other DRDA Application Servers", "Referential integrity" and "Automating the large copy load" sections. Added a new major section "Tools". Revised "Outer join of two or more source tables with common primary key" section. Minor edits to "Two-way propagation" section. Renamed "Copying from DB2/2 or DB2/6000 to DB2 or DB2/400" to "To PUSH or to PULL", with changed text and example setups using DPRTOOLS. Added additional text in "Work-around for copy-owner naming restriction" section. Dropped the "Triggering DB2 Application Code", "To register a source user table", "To capture changes to the source table", "To subscribe to the source table" sections, as the new "Using DPRTOOLS in a data warehouse view scenario" section on page 46 illustrates these points more precisely. Rewrite the "When you've added new columns to a user table" section to include a major example scenario. Add "Transactions which will not propagate correctly" and "Using DPRTOOLS in a data warehouse union scenario" sections. Dropped the "Centralized control/batch copy definition" section. Added the practical "Migrating from 'test' to 'production'" section. Added use of INTERVAL_MINUTES=0 to "Controlling the *timing* of copying" section. Rewrote the "Database Partitioning" example to use DPRTOOLS instead of JOINREG and UNISUB. Added recommendation to use DPRTOOLS in the "Propagating to DB2/6000 Parallel Edition" section. Removed the section "The tools JOINREG, UNISUB and UNION" and all references to these tools. Replaced sample query in "Determining change capture program lag", which was in error.

Abstract

This paper is for new users of DataPropagator Relational who desire tips and hints on how to tailor the product to meet specific application requirements. After using this document, please recommend which, if any, of these tips should be included in the official product documentation. IBM employees are welcome to post comments in the DPROPR FORUM on the IBMPC conference disk. Customers may post comments and questions directly to the DPROPR CFORUM through IBM's *Talklink* facility.

You are also invited to contact the author directly to report errors in this guide:

Rob Goldring
telephone: (408) 463-2316
VNET: GOLDRING at STLVM14
Internet: goldring@vnet.ibm.com

Note: some of these procedures have not been tested. Please validate any procedures of interest and kindly let us know of any errors in this document.

Note: references to the IBMSNAP_* control tables (as distinct from the IBMSNAP_* columns present in CD and copy tables) in your subscriptions, etc., creating version dependencies in your installation which may burden your migration to a future version of the product.

The following IBM and NatWest employees designed, developed, tested and documented this fine product for you:

Tadakatsu Azuma, Lin Chang, Wes Cheng, Ho Choi, Rob Goldring, Beth Hamel, Pete Harrison, Madhu Kochar, Peter Lien, Elma Lui, Cecile Madsen, Chris McCarthy, Andy Pearse, Jane Prusaitis, Holly Rader, Dave Reisner, Gerda Sander, Sandy Sherrill, Pat Shimer, Geoff Shuller, Shawn Swanson, Billy Taneja, Mark Turner, Shu-Jen Wang, Julie Wood, Percy Yu, Mike Zampiceni, Sergey Zundel

Contents

Introduction

To new users, DataPropagator Relational is a copy management product with solid, basic capability and a nice graphical user interface. To those knowledgeable of its distributed SQL control table design, DataPropagator Relational is a tool of immense flexibility, able to efficiently maintain copies small or huge, defined as simple subsets of single tables or complex joins and unions of many tables, histories of coarse or atomic granularity, over dedicated LANs, WANs or dial-up telephone links, and among databases on everything from mainframe to notebook computers.

Moving beyond the "solid, basic" capabilities of the product *does* require some mental investment. This application guide is intended to give you a head start into the known advanced techniques. Some of the theory behind what you will read in this guide is complex, so don't feel badly if a few of the suggested techniques fly right over your head the first time you read of them. We in the development team, together with our early customers, have been learning and growing in our understanding of data replication every day. Should you choose to become a product expert, *you will not become bored!*

Nothing is more confusing to new users of the DPropR/2 administration component than the security model (nothing that is, except for setting up the network). Be sure you take time to understand the security model before you begin using DPropR/2. Those who begin (trying to) using the product before understanding the security scheme usually become quickly frustrated. Oh, and by the way, the administration security scheme is due for a major overhaul and simplification, based on feedback we've received. Many of the examples in this guide are shown using the DPRTOOLS administration interface, a prototype user interface without a rigid security model.

DPropR/2 administration is defined in terms of three user roles: a single *administrator*, few *registrars* and many *subscribers*. Registrars will typically be database administrators, while subscribers are in the class of users who are able to query data but typically lack administrative authority. Registrars are assumed to have more data processing knowledge than subscribers and perform the more technical tasks for the benefit of subscribers.

Customers, unfortunately for both themselves and our telephone support staff, routinely make the following errors in judgement:

1. they begin a large client/server project without inviting a networking expert onto the project team, or,

2. they don't even think to do the data architecture work in advance, with project delays the most common result.

Why Copy?

Leading consultant Colin White lists six reasons for copying data:[WHITE92]

- avoid disruption to the operational system by having users query copies, not the operational data directly
- performance
- data security
- different user views
- data currency
- data cleanup

System Structure

DataPropagator Relational is comprised of three semi-autonomous program components, shown in grey in Figure 1. There are no standard program linkages between these distributed components -- all control data and user data flows between them via SQL tables. None of these components then, has any dependency on any particular type of physical network, network protocol, communications API, or any of that. Furthermore, you can substitute your own program for one (or more) of the components.

Any of the components is able to continue, regardless of the state of the other components. All state and change data is stored in stable SQL storage, so user data is not lost as a result of a program failure. As all definitions are in SQL tables, they may be changed while the system is running. All three DataPropagator Relational components are designed for availability. Multithreading, data communications, and data integrity support are provided by the underlying SQL DBMS engines. Typically, copies are maintained without routine access to your application tables. The workload of capturing changes and propagating them to other sites can be spread throughout the day, so that there is no need to consume valuable *batch window* time, as is true

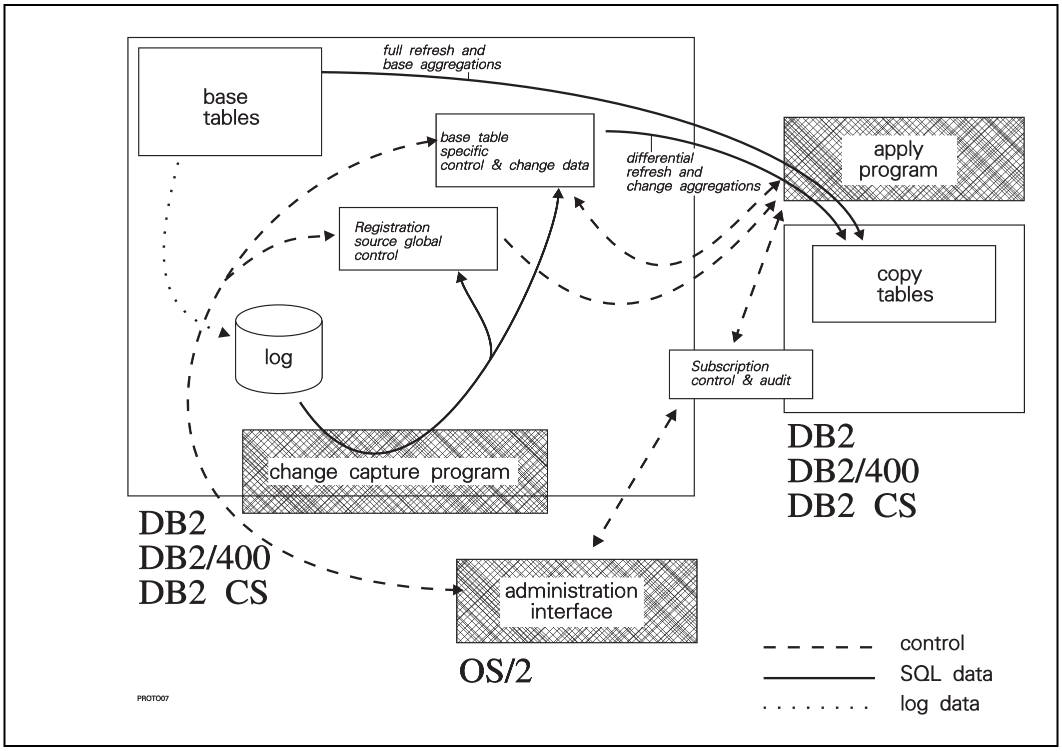

Figure 1 Abstract system structure showing three components.

when using simple data extract products.

Registration

Tables must be *registered* to DataPropagator Relational before they are eligible to be copied. There are good reasons for this:

- Security. With base table registration, copying is made a controlled activity. Copying privileges are assumed to be held by only a subset of those with direct access privileges. Copy privileges, a collection of specific SQL privileges, are enforced by the SQL authorization facilities.

- Source system impact. Full-refresh is enabled or disabled for a given base table, separately from differential-refresh activity, determined by the *fullrefreshenable* control flag in the IBMSNAP_CD_CNTL registration table. The *fullrefreshenable* flag for the base table must be turned on for new *complete* copies to be initialized.

- Bookkeeping. Each registered base table has an associated *pruning control table*. Used internally by DataPropagator Relational to signal when change data can be discarded, this table is also a useful external record giving the name and time of last refresh for every subscription.

The features listed above offer the access control, performance governing, and monitoring capability necessary to manage a large scale copy management system.

A *registrar* of a given base table is the administrator controlling the copying of the table and the allocation of resources for holding the captured changes to the table. Two DataHub display object types are of interest to registrars -- *(DPropR) Tables to be registered* and *(DPropR) Tables - registered*. Registrars initiate the granting of specific privileges required by subscribers.

The ASN.IBMSNAP_CD_CNTL table contains the list of registered sources. The rows in this table tell the Capture component which base tables should have changes captured, and into which change data tables.

Subscription

Analogous to automatic delivery of a newspaper or other periodical, *subscription* to a base table (and implicitly, any change data or consistent change data table included in the base table registration) implies automatic, periodic delivery of current data from the base table. Any number of subscribers can define any number of subscriptions for a given base table.

Any user making copies is a *subscriber*. The subscriber user ID may in fact represent an individual, or a logical user, such as a departmental work group.

Two DataHub display object types are of interest to subscribers -- *(DPropR) Subscriptions to be defined* and *(DPropR) Subscriptions - defined*. Subscription sources are all the base tables available to the subscriber for copying (i.e., the sources are registered and the subscriber has sufficient privileges to access and copy from the tables). Current subscriptions are all the copy requests already defined to DataPropagator Relational.

Point in Time and Aggregate Copies

IBM Research defined data base "snapshots" in the 1980's as copies automatically maintained using *full refresh* and/or *differential refresh* algorithms. DataPropagator Relational calls the classic snapshot a *point in time* copy and defines a specific table structure. DataPropagator Relational time stamps every row in a point in time copy to indicate data currency. The exception to this is DPropR/400, which is the first DPropR implementation to support *user copies*, the new name for copies which contain no additional IBMSNAP_* columns.

Aggregates contain the results of SQL column function calculations, *appended* at each re-calculation, so that aggregates build up a history over time. Base aggregates are calculated directly from a user table or point in time copy. An example of a base aggregate would be the daily counting of all inventory items. Each calculation represented in a base aggregate is stamped with the time of calculation.

Change aggregates are calculated from the change data related to the user table. A table with 1,000,000 rows might only have 1,000 recent changes, so a change aggregate is much more selective than a base aggregate. A change aggregate might represent weekly sales totals. Each week, a new result is calculated and appended to the aggregate, stamped with a beginning and ending time stamp to indicate the weekly interval of the sales total calculation.

Change aggregation may be the single most powerful feature of DataPropagator Relational. Along with the power and flexibility comes the burden on the user to specify the change aggregate's subscription predicates correctly -- the *where* clause. A change aggregation is calculated each time over an interval which may include any number of *insert*, *update* and *delete* operations. Consider these three change operations when defining your change aggregate subscription.

For example, say that you want to track the number of new customers added to the CUSTOMER table each day. We'll assume that you are already capturing changes for this table. To set up the subscription, you would start by selecting the CUSTOMER table from the list of *(DPropR) Subscriptions to be defined*. Within the subscription *notebook*, click on the columns page and enter **COUNT(*)** as an SQL expression and name the target column **NEW_CUSTOMERS**. Then click on the frequency page and set the refresh interval to daily. Then click on the rows page, and set the WHERE clause to **IBMSNAP_OPERATION=''I''**, so that only the inserts are counted. Without the predicate, the calculation would include updates to, and deletions from, the existing CUSTOMER entries -- not the intended result. In fact, it might be interesting to track all three change operations separately.

Staging Tables

Staging tables, properly named *consistent change data* or simply CCD tables, are useful for saving complete histories, for designing generalized "information warehouses" and in general for supporting a variety of data delivery configurations.

Changes can easily be propagated from other systems via staging tables. For example, IBM's IMS Data Propagator (DProp) product optionally delivers IMS change data into a DataPropagator Relational *Consistent Change Data* (CCD) staging table, which can then be registered as a source for subsequent copying within the DataPropagator Relational copy management network.

Staging tables can be local or remote from the original source, and be defined as *complete* (or not) and *condensed* (or not). *Complete* means that the table contains every row of interest from the original source. If a table is not complete, it cannot be used as a source to initialize point in time copies. *Condensed* means that only the latest value for the row is present, non-condensed CCD tables contain complete histories. Figure 2 gives a quick reference guide understanding how the *complete* and *condensed* attributes describe the contents of CCD staging tables.

CCD local/remote	CCD complete	CCD condensed	where used
local to original user table	Y (must be set outside of the Administration Facility)	Y	Probably doesn't make sense as the CCD table is redundant with the local user table and not usable as a history
local to remote copies	Y (default)	Y	Remote staging table, where history retention is not a requirement
local to original user table	N (default)	Y	Efficient local staging, providing a stable source for effecting synchronization of "fan out" copies. Using a CCD staging table also relieves the CD table from lock contention problems when many apply programs are attempting to read from the CD table while the change capture program is writing to the CD table.
local to remote copies	N (must be set outside of the Administration Facility)	Y	Risky, as cannot support initialization of new point in time copies. Used by experts who need to write their own custom apply programs, such as to maintain indexed files or foreign DBMS tables.
local to original user table	Y (must be set outside of the Administration Facility)	N	Local history.
local to remote copies	Y (default)	N	Remote history.
local to original user table	N (default)	N	General purpose local staging table, can, together with the original user table, support copies which are complete histories. This table has all change data over a given interval, but is not a complete history without the user table.
local to remote copies	N (must be set outside of the Administration Facility)	N	Risky, as cannot support initialization of point in time copies. Used by experts with application requirements for change data only.

Figure 2 Guide to defining consistent change data (CCD) staging tables.

Figure 3 offers advice on a few of the practical benefits of using staging tables to deliver data efficiently, maintain histories, or both.

You can even customize the granularity of history to save. Say you would like not to save each and every change, a *non-condensed* history, but you would like to save the last value for each row changed on a given day. To accomplish this, simply *create a column* using the DATE function applied to the IBMSNAP_LOGMARKER timestamp column and include this computed column in the primary key of a *condensed* CCD staging table. New rows in this table will be updated in place for one day, then new rows will be added the next day and these in turn will be updated in place for just one day. This is just one of many possible "in between" history-keeping techniques. Be creative!

application requirement	recommendation
"I have many requests for copying data from DB2 MVS to DB2/2 LAN servers. Many of the requested copies are very similar -- have data in common, but differ slightly in their subsetting or currency requirements (some users need updates ASAP, while others want only daily or weekly updates)."	Place a *complete* staging table in the DB2/2 data base. Make it *condensed* and subset the rows and columns just enough to satisfy the data requirements of all, while still saving as much on storage as possible. You'll be able to set up all the definitions using the DPropR administration facility. Set up the individual copies to be locally maintained from the DB2/2 staging table. By using a *complete* staging table, new copies may be defined and initialized at any time, without need to access the DB2 MVS system. Only the staging table is maintained directly from the DB2 system. Each data element moves across the wide area network just *once*.
"Our auditors are asking us to find a way to maintain and support on-line access to a great deal of historical data."	The administration facility could set up a basic definition for you, but for large DB2 MVS tables you will want to manually define your staging table to take advantage of partitioned tablespaces. You will also want to create indexes tailored to your anticipated query requirements.
"We have some groups which are housed in different buildings, with autonomous LANs, yet want to be sure they share the same "view" of the primary data base reflected in their copies."	Three techniques, used together will facilitate the cross-site consistency needed: • if groups of tables need to be copied from a single source with referential integrity, create staging tables at the source with definitions to coordinate the maintenance of the staging tables (see page 30) • carefully control the frequency, and possibly the time, of when the staging tables are updated (see page ?) to introduce stability among the staging tables. • set the frequency of update for the remote copies to be often enough for the copies to "settle down" within the desired period. For example, if the staging tables were updated sometime after 6:00 each evening, a refresh frequency of 4 hours would mean that, on average, the remote copies would be converging to a new state about 8:00pm and the remaining copies would begin converging by 10:00pm. When the workday begins the next morning, all the copies would be consistent across sites and remain stable until 6:00pm the following day.

Figure 3 Recommendations on how to use staging table to satisfy some common application requirements.

The examples in Figure 4, Figure 5 and Figure 6 will help illustrate when staging is most useful (and maybe even when it is not). Data at a central DB2 for MVS server, MVS1, is being propagated to point in time copies on five DB2 workstation servers, WKS1..WKS5. Figure 4 shows the use of a single CCD table at the data server (source), dubbed an *internal CCD* because it is hidden as an alternative source for base table base1, just as change data table CD1 is a hidden, alternative source table. The attribute CCD_COMPLETE='N' is the default used for internal CCDs by DPropR/2 and the attributed CCD_CONDENSED='Y' is prompted at the closing of the subscription notebook when defining the subscription for the internal CCD. This is the best choice for reducing the number of rows that will flow over the network.

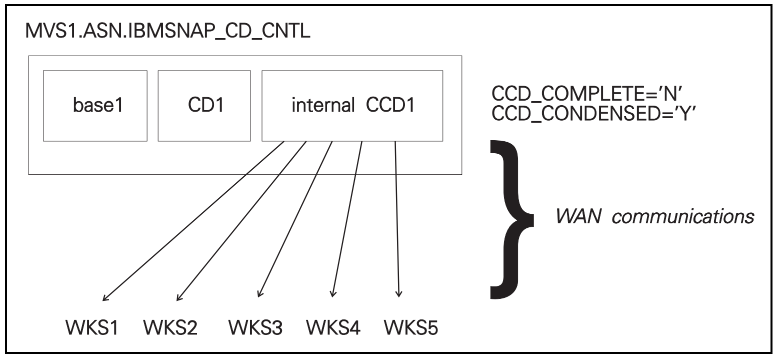

Figure 4 Use of "internal" CCD table, for staging at the data server, to reduce join processing costs.

In this example, the change data rows from CD1 need only be joined with the IBMSNAP_UOW table *once*, when updating the internal CCD table. The subscriptions for the point in time copies at the five workstation servers will ordinarily choose the internal CCD table as the physical source table, avoiding CD1, and thus avoiding the cost of additional joins with the UOW table. The disk space consumed by the internal CCD table will serve to reduce subsequent CPU costs for subscription processing.

Note: don't forget to prune the internal CCD table, in order to keep its size to a minimum.
>GING2Fi

In Figure 5, the CCD table is remote, so needs to be *complete*, the default chosen by DPropR/2, to be able to initialize the point in time copies at the workstation servers. The copies on the workstation servers then are isolated from the data on MVS1 -- all maintenance of these copies is via the base CCD table on WKS1.

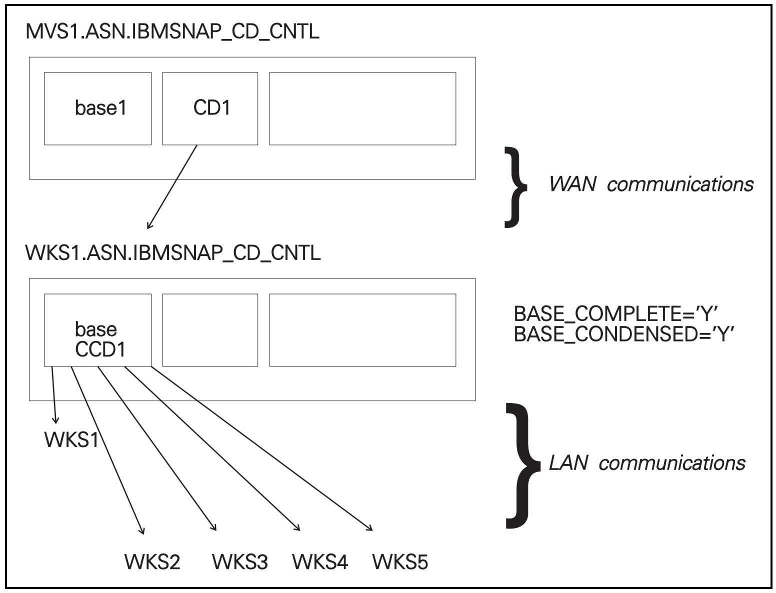

Figure 5 Using a remote CCD, to reduce WAN communications.

With this configuration, a change data row at MVS1 need flow over the wide area network (WAN) at most once, to the CCD table on WKS1. Note that this provides a join optimization similar to Figure 4. From the CCD table on WKS1, each change data row will flow at least one more time, to one or more copies on the workstation servers, but always over the local area network (LAN) and not over the more expensive WAN.

The remote CCD table, being complete, is going to require more space than the non-complete internal staging table in Figure 4. The benefit derived from this additional investment in disk space is that the central database MVS1 only sees a 1-subscription workload, not a 5-subscription workload. Even if the remote staging table on WKS1 were supporting 20 other servers instead of just 5, MVS1 would still only see a 1-subscription workload.

Unlike the internal CCD in Figure 4, the complete, condensed staging table in Figure 5 requires no pruning. When making your space estimates, don't forget to allow for the *deleted* rows which will remain in the CCD table, as there is no such thing as destructive deletes against CCD tables, only *inserts* and *updates*.

Figure 6 shows both the internal staging of Figure 4 and the remote staging of Figure 5. As the configuration in Figure 5 already reduced the join processing load on MVS1, there is no additional join processing savings gained by having the internal CCD on MVS1; in either case the join is performed just once.

There is however, the potential for a network optimization, though one more subtle than gained by the use of remote stating in Figure 5. The savings here would come from eliminating "hot spot" *duplicates* before propagation from MVS1. If CD1 is used as the physical source table, there is the potential for propagating more than one row per key, as change data tables by definition are *not condensed*.

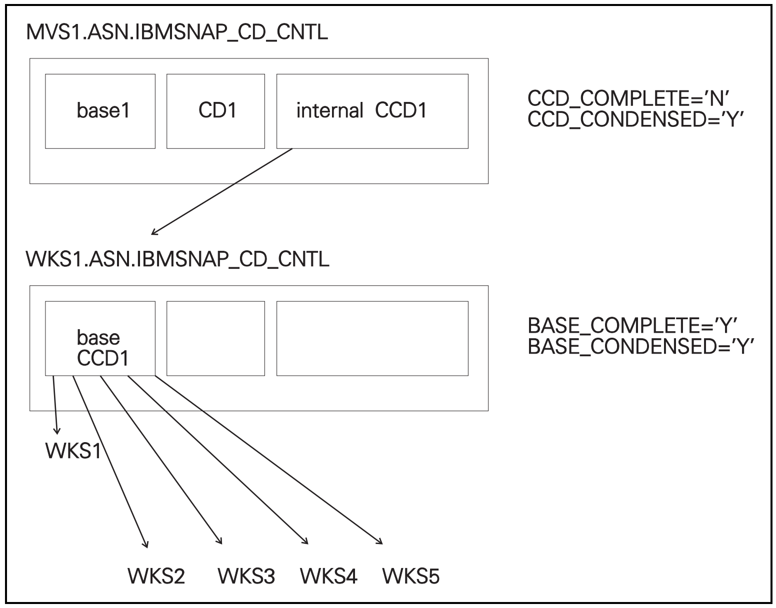

Figure 6 Use of both internal and remote staging, offering network transmission savings in *hot spot* update scenarios.

An even more subtle consideration you might take into account is that the use internal staging (Figure 4 and Figure 6) results in the CD+UOW join being done locally to the target table, the internal CCD. In the remote staging case in Figure 5, the CD+UOW join is done remotely from the target, so requires the availability of a network connection. Use of internal staging then, can reduce the duration of the remote sessions, by separating the CD+UOW join work from the elapsed time of the remote propagation.

Your decision then on whether to use both internal staging and remote staging will need to consider the tradeoffs of the extra space, relative to these potential, and subtle network savings. If the likelihood of duplicates is low, and the number of copies (potential for redundant join processing) is low, you may or may not consider the internal CCD in Figure 6 as simply adding cost, in terms of both space and staging delay.

Where to place subscription control tables?

The question of where to place control information, whether to designate the *control server* as being either the *data server* or *copy server*, seems to be determined by the personality of the organization. Centrally managed companies seem to want control information to be consolidated in one or few places, while decentralized companies insist on each business unit having its own, private, control store. The preferences seemed to be based more on corporate culture than on technical reasoning. This leads me to believe that there is no one "best" solution. For those of you out in the field talking to many customers, do not be surprised if customer B wants a control table placement scheme just the opposite of what customer A wanted. For what it's worth, that's my 30,000 foot observation.

Now, on the technical side, the general case is certainly to allow the customer to place control information anywhere they like. This is the academically "correct" thing to do, is certainly expected by techies, but causes other problems. Flexibility brings complexity, so choosing an appropriate set of design restrictions can make a product easier to use. You hopefully agree that allowing the customer to choose the placement of each and every control table (and there may be hundreds) would be too much flexibility for the system to appear easy to use. There would be so many decisions to ponder.

The Capture/x implementations are not distributed at all, so registration information must be local. This makes sense, as registration information is logically an extension of the local DBMS catalog. The Apply/x implementations though, can tolerate their subscription control tables being placed anywhere in the network. The restrictions on subscription-related control table placement come from the DPropR/2 administration component, based on conscious choices to take the general flexibility supported by Apply/x and reduce perceived complexity while retaining an essential degree of flexibility.

Next, I'll talk about three broad control table placement strategies and why we implemented the two in the DPropR/2 user interface that we did.

COPY SERVER PLACEMENT. This is for organizations that favor decentralized management. The IBMSNAP_REF_CNTL table contains lots of settings, for example the time value describing the refresh interval. Ownership of this control information may be an issue. If this important control value is owned by "me" and placed in "my" database, then "you" cannot change it. "You" in this case would be a system administrator at the source site; a user with authority to access tables used by my apply program, without my consent or knowledge. If "I" want to retain control, I will choose to place the control information in a database where "you" cannot get to it.

Remember, database administrators might very well want to gain access to subscription definitions, to disable them, to rearrange their priority, to change their frequency of update, etc. Some customers are adamant that they do not want to surrender their subscription definitions to any of "those people" who's data they are copying. Copy server placement is for these customers.

Copy server placement also has a number of performance advantages which justify its general recommendation.

1) Each Apply program will refer to its control server or servers frequently, to check to see if there are any subscriptions requiring service. Such schedule analysis will be cheaper if these control tables are locally accessible to the Apply program.

2) Much of the Apply overhead costs can be shifted away from a popular data source by placing the subscription control tables elsewhere.

3) Even when doing remote work, it is slightly more efficient to have the subscription control tables within the same connection-scope as the copies.

DATA SERVER PLACEMENT. This is for centralized organizations. Here, the perspective that of the source database administrator. He/she can see all of the subscriptions for tables in databases for which they have responsibility, and can gain access to all these definitions to modify them, should they choose to do so. For ease of future migrations, use this option when there is only *one* data server per Apply process. Note: this option can range from slightly more expensive than copy server placement to *greatly* more expensive if there are many cycles taken up merely polling the IBMSNAP_REF_CNTL table for work when there are, in fact, no eligible subscriptions. In this case, there will be many useless connections made to the (assumed remote) data server, resulting in communications costs but no real work being accomplished.

CENTRALIZED PLACEMENT. We considered this, but decided to drop support for it early in our administration GUI design discussions. From a reliability and availability perspective, this is the worst option, as it introduces a single point of failure. If the single control server is disabled, all of the apply programs, wherever they are running, are prevented from processing subscriptions. For the two options above, any single database can be lost and only partially disable the replication network. Loss or unavailability of a single central control database would be catastrophic, as no copies could be updated, anywhere, until access to the central control database was restored. For reporting and analysis purposes, one could certainly choose to define subscriptions for the control tables, and place these copies in a single database.

Still, some customers prefer to centralize control information for ease of management. You can use the DPRTOOLS to define and maintain a single-control server configuration (see the "Tools" section beginning on page 36).

Pictorial examples of control server options

DPropR uses four logical server names to describe the placement of data and control tables. The *data server* is the RDB where the source table is found. *Data server* is a term borrowed from the DRDA terminology. The *copy server* is the RDB where the copy is to be maintained. The *control server* is the RDB where the subscription-related control information is stored. The *local server* is the RDB that is the default RDB for an Apply/x program. The *local server* is where Apply/x expects to find its routing table, IBMSNAP_ROUTING. The routing table lists the names of one or more *control servers* where the Apply/x program will find its private subscription definitions. We'll use the terms *data server*, *copy server*, *control server* and *local server* in the following examples.

The DPpropR/2 product, which supports DPropR administration from the graphical DataHub/2 control point interface, understands two *control server* placement options. The customer can choose either of the two conventions on an individual subscriber basis:

1. the *data server* is used as the *control server*, or
2. the *copy server* is used as the *control server*.

In order to more clearly show the results of choosing one control server option over another, we'll use the sample configuration in Figure 7. The configuration shown in Figure 7 contains two source tables, SOURCE_A and SOURCE_B, which are stored at RDB_A and RDB_B respectively. For the purposes of these examples, assume that *RDB_A*, *RDB_B*, *RDB_C*, *RDB_D* and *RDB_E* could be any member of the DB2 family: DB2, DB2/400, DB2/6000, DB2/2, etc.. We'll assume that the customer's requirement is for DPropR to maintain point in time copies (PIT_A, PIT_B) of each of these two source tables (BASE_A, BASE_B) in the *copy servers* RDB_D and RDB_E. Additionally, the propagation of updates from RDB_A and RDB_B are to be "staged" in CCD tables in RDB_C. The point in time copies then, are maintained via the CCD tables in RDB_C.

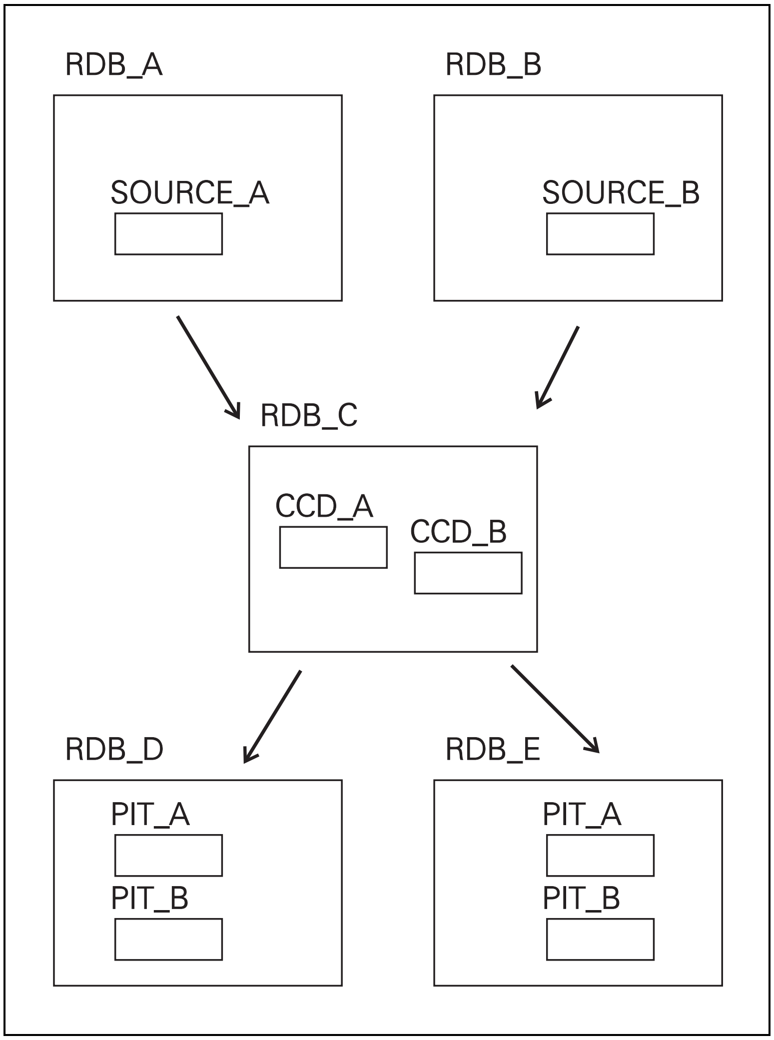

Figure 7 Sample configuration, showing only the source tables, intermediate staging and point in time copy tables.

Figure 8 introduces the apply programs to be installed and run at RDB_C, RDB_D and RDB_E. Each Apply/x program is associated with a subscriber's user ID and will execute using that subscriber's user ID. The Apply/x execution at RDB_C is associated with user SUBc, the Apply/x execution at RDB_D is associated with user SUBd and the Apply/x execution at RDB_E is associated with SUBe.

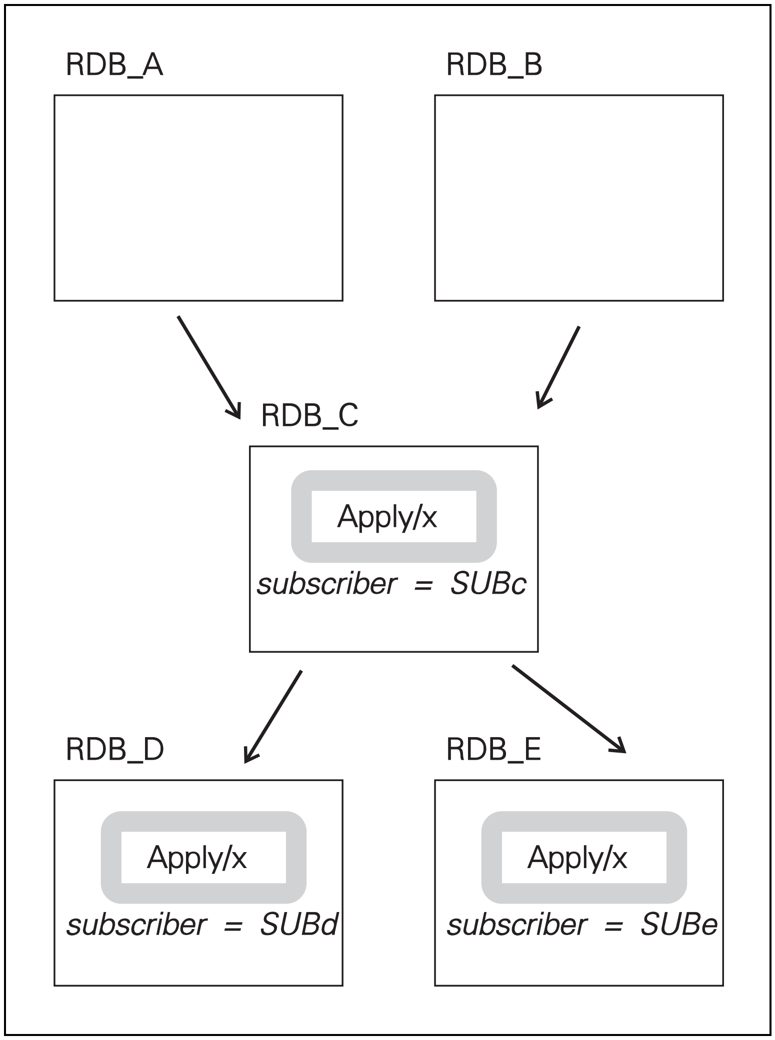

Figure 8 Sample configuration from Figure 7, illustrating Apply program placement.

Figure 9 introduces the routing, refresh control, refresh columns and audit trail control tables owned by each of the subscribers. Figure 9 shows how these tables would be placed, and the *control server* names that would appear in the IBMSNAP_ROUTING tables, assuming that the *control server* = *data server* option was chosen by subscribers SUBc, SUBd and SUBe. Tip: for ease of future migration, subscriber SUBc should avoid using this option as there is more than one data server referenced.

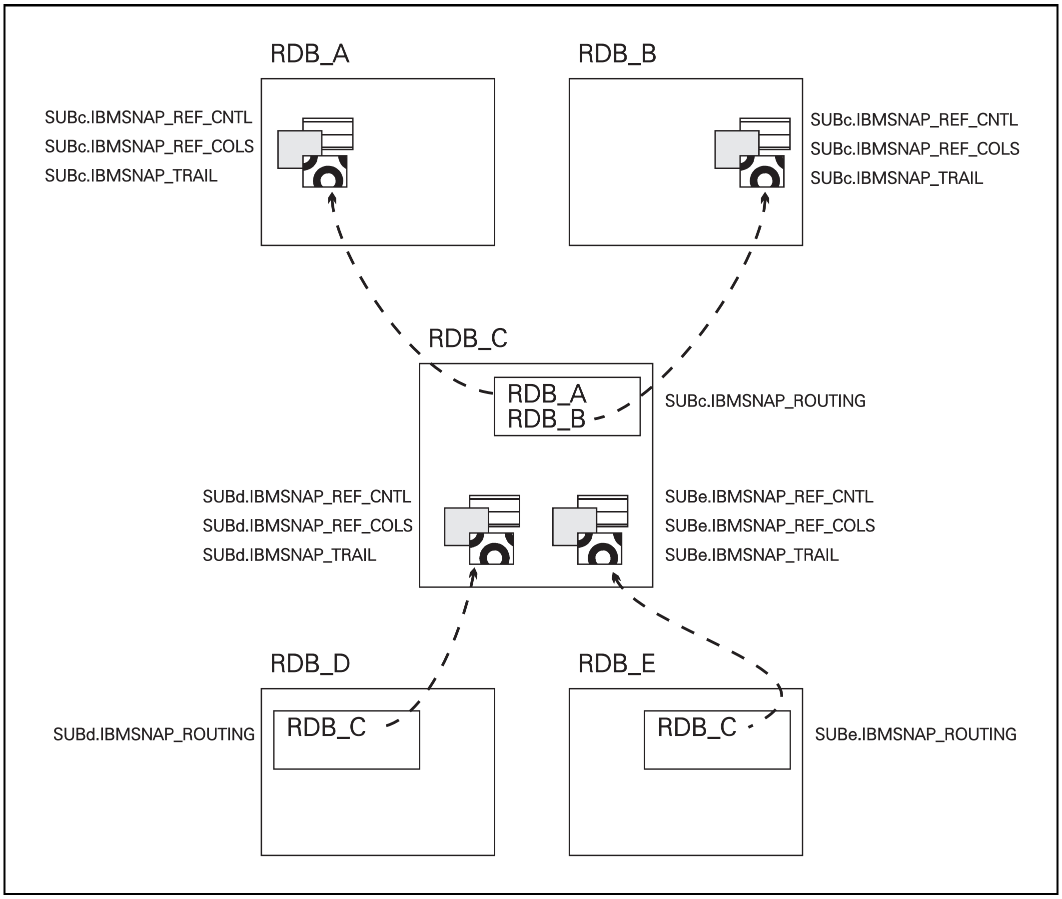

Figure 9 Sample configuration from Figure 7, with *control server* = *data server*.

Figure 10 shows how these tables would be placed, and the *control server* names that would appear in the IBMSNAP_ROUTING tables, assuming that the *control server* = *copy server* option was used throughout.

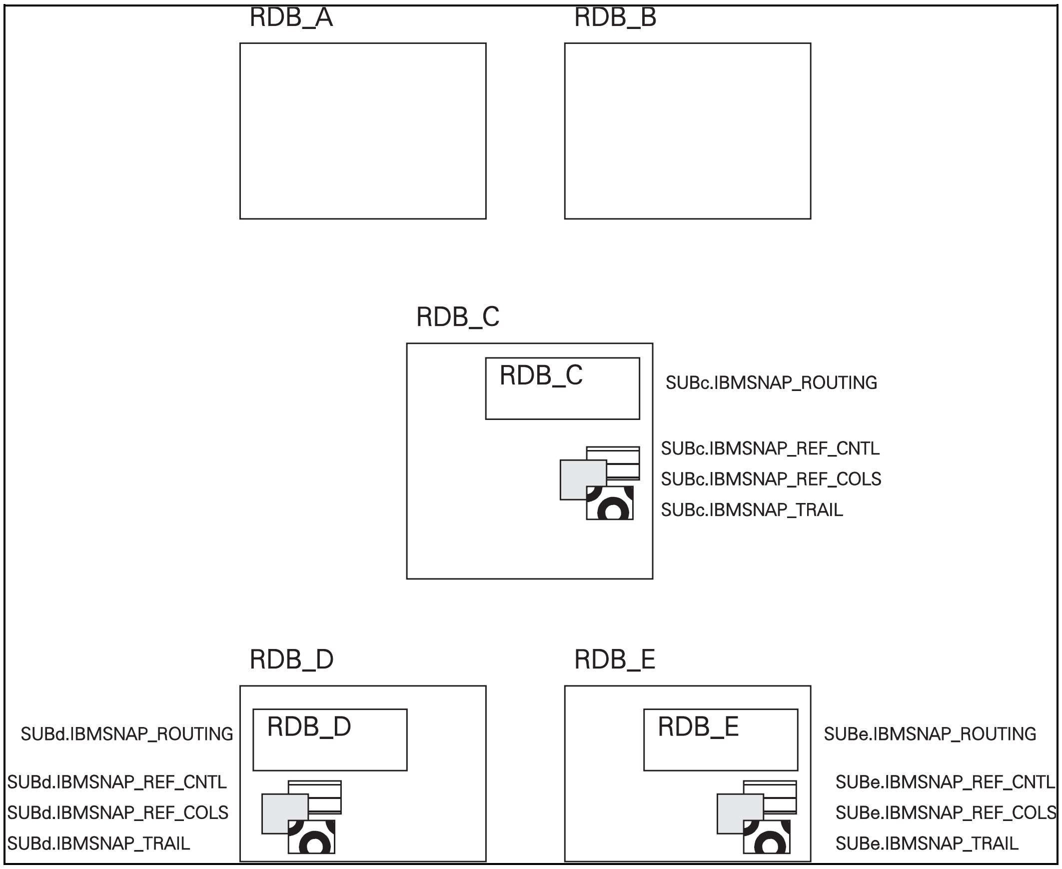

Figure 10 Sample configuration from Figure 7, with *control server* = *copy server*.

Figure 11 shows how these tables would be placed, and the *control server* names that would appear in the IBMSNAP_ROUTING tables, in a custom configuration assuming that RDB_A were designated as the sole, central *control server*. The only tables shown in Figure 11 are the control tables owned by subscribers SUBc, SUBd and SUBe. Such a configuration is not supported using DPropR/2, but can be supported by Apply/x using DPRTOOLS (see page 36). DPropR/2 can be thought of as a fancy SQL editor, one which creates copy tables and control tables and maintains definitions in control tables. As such, any task performed by DPropR/2 can always be substituted with direct SQL processing, or with an alternative tool, such as DPRTOOLS.

This type of configuration then, leads to interesting possibilities for maintaining a replication network of indefinite size using centralized batch definition techniques. Still, to avoid the single point of failure associated with the notion of a central *control server*, one should consider the technique shown in Figure 10.

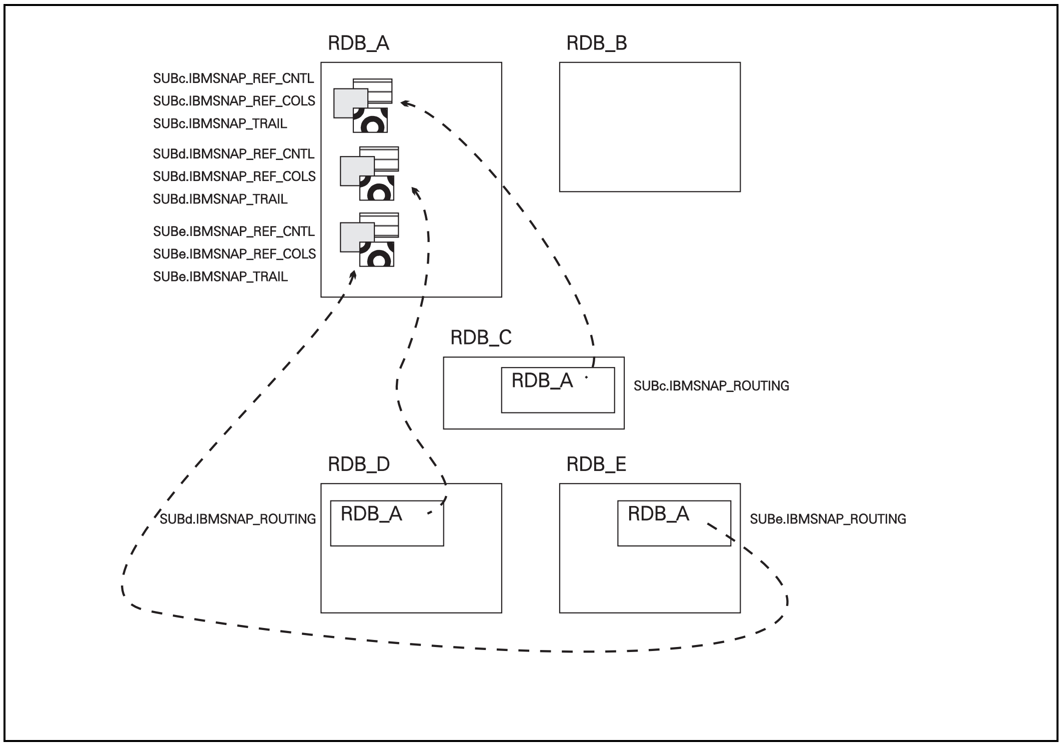

Figure 11 Sample configuration from Figure 7, with custom installation having RDB_A designated as a centralized *control server*.

Using Extract Files

DataPropagator Relational is able to propagate changes, even changes from non-relational sources. In those cases where only extract files are available, net changes between old and new extracts can be calculated and maintained in CCD staging tables.

```
CREATE TABLE CCD_TABLE (
  IBMSNAP_LOGMARKER TIMESTAMP NOT NULL, /* approx time of change */
  IBMSNAP_INTENTSEQ CHAR(10) NOT NULL,  /* sequence within trx.  */
  IBMSNAP_COMMITSEQ CHAR(10) NOT NULL,  /* trx. sequence         */
  IBMSNAP_OPERATION CHAR(1) NOT NULL, /* I-insert,U-update,D-delete */
  KEY1 CHAR(1) NOT NULL,      /* user data */
  DATA1 CHAR(1),              /* user data */
  DATA2 SMALLINT);            /* user data */

INSERT INTO CCD_TABLE (
  IBMSNAP_LOGMARKER,
  IBMSNAP_INTENTSEQ,
  IBMSNAP_COMMITSEQ,
  IBMSNAP_OPERATION,
  KEY1,
  DATA1,
  DATA2)  /* calculate logical deletes */
        SELECT    CURRENT    TIMESTAMP,    X'00000000000000000000',
        CHAR(DATE(CURRENT TIMESTAMP)),'D', X.KEY1, X.DATA1,X.DATA2 FROM
        CCD_TABLE X WHERE NOT EXISTS
                  (SELECT * FROM TEMP WHERE X.DATA1=TEMP.DATA1     AND
                  X.DATA2=TEMP.DATA2 AND  X.KEY1 =TEMP.KEY1);

INSERT INTO CCD_TABLE (
  IBMSNAP_LOGMARKER,
  IBMSNAP_INTENTSEQ,
  IBMSNAP_COMMITSEQ,
  IBMSNAP_OPERATION,
  KEY1,
  DATA1,
  DATA2)  /* calculate logical inserts */
        SELECT    CURRENT    TIMESTAMP,    X'00000000000000000001',
        CHAR(DATE(CURRENT  TIMESTAMP)),  'I',  TEMP.DATA1,  TEMP.DATA2,
        TEMP,KEY1 FROM TEMP WHERE NOT EXISTS
                  (SELECT * FROM CCD_TABLE Y WHERE Y.DATA1=TEMP.DATA1
                  AND  Y.DATA2=TEMP.DATA2 AND  Y.KEY1 =TEMP.KEY1);

COMMIT;
```

Figure 12 Using SQL expressions to maintain *complete*, *partially condensed* staging table CCD_TABLE from extract files loaded into table TEMP.

One method of maintaining change data from extract files is to always load the new extract into a temporary table and use SQL expressions to calculate the differences between the data now in the CCD table and the new extract data. Figure 12 shows a very simplistic approach to maintaining a partial history (assuming "net" changes computed from daily extracts, the second by second time series detail is unknown). In this simplistic approach to calculating differences, updated rows are represented by a deleted row followed by an inserted row.

Performance tips

The number 1 tip: if you're an MVS customer, stop using release 1 code and be sure to have the NOPRUNE PTF installed so that you can use the NOPRUNE Capture option. Customers have reported a 91% CPU reduction (and a complementary 50% CPU reduction for Apply processing, due to reduced contention) using this option and pruning just once a day.

For DB2 change data and UOW tables, try using segmented tablespaces with table level locking. Be sure to set the catalog statistics to indicate high index selectivity. If you're creating your own DB2 MVS tablespaces, specify

CLOSE=NO option when creating tablespaces and indexes for the UOW and CD tables. Some customers report 50% Capture/MVS CPU cost reductions with this one tuning tip. This change avoids the performance problem with DB2 Version 2.3 *slow close* processing described in APAR PN45256.

Performance will likely improve if you maintain two indexes on the ASN.IBMSNAP_UOW table, one to support fast apply processing and the other to support fast pruning. If you use DPRTOOLS to define your environment, these are the indexes you'll get. Taken together, the indexes can significantly reduce your overall Capture CPU costs, as much as a 50% reduction, as measured by a large MVS customer.

UOW table indexes

for fetch performance:

```
CREATE UNIQUE INDEX U1yymmddhhmmssIX1 ON ASN.IBMSNAP_UOW
(IBMSNAP_COMMITSEQ ASC, IBMSNAP_UOWID ASC, IBMSNAP_LOGMARKER
ASC)
```

for pruning performance:

If MVS then:

```
CREATE UNIQUE INDEX U2yymmddhhmmssIX2 ON ASN.IBMSNAP_UOW
(IBMSNAP_COMMITSEQ DESC, IBMSNAP_UOWID ASC)
```

Otherwise then:

```
CREATE UNIQUE INDEX U2yymmddhhmmssIX2 ON ASN.IBMSNAP_UOW
(IBMSNAP_UOWID DESC)
```

CD table index

If on MVS or AS/400 then:

```
CREATE INDEX CDyymmddhhmmssIX ON cd (IBMSNAP_UOWID ASC);
```

Otherwise then:

```
CREATE INDEX CDyymmddhhmmssIX ON cd (IBMSNAP_UOWID DESC);
```

Try using a longer commit interval for Capture/MVS, batching more inserts into one unit of work (if you are using an interval of 30 seconds, try 60 seconds, etc.). This reduces both dynamic bind and automatic pruning activity, should you not be using the NOPRUNE option. Turn off your Apply programs during the day, choosing to defer data communications, table update costs and change data pruning costs until off-hours.

Typically, you will leave Capture/MVS running continuously, but you may consider suspending Capture/MVS when running DB2 utilities. For example, if the tablespace you are trying to image copy contains a change data table, Capture/MVS may attempt to insert into the table and will be blocked by the utility. The simplest protection then, is to suspend Capture/MVS during your utility batch window.

When the load on Capture is especially large, such as when a batch application is updating millions of records at a time, you may want to stop the Apply processes, allowing the Capture process to run without possibility of lock contention on the critical section table.

One large customer recommends reorganizing CD tables in DB2 on a weekly basis.

Concerns for customers with Apply/MVS 1.2.1

The initial distribution of Apply/MVS 1.2.1 contains a new feature that automatically advances LAST_DIFF_SEQUENCE and PRUNE_LIMIT_SEQ, even in cases where no rows qualify to be applied to the copy. This change averts the problem described in the "When pruning doesn't seem to be effective" section on page 23.

Unfortunately, this feature, by advancing these control values regardless of what has been applied, invalidate techniques that rely on these columns, such as those explained in the "Referential integrity" section on page 30 and the "Protecting against an abnormally large apply volume" section below. There is a hyper-PTF (as of April 14, 1996 not yet available) to make the new feature optional, on a subscription by subscription basis, allowing customers to continue using the customization techniques mentioned.

This notice is of no concern for users of Apply products on other platforms; platforms other than MVS. Any other implementations of Apply 1.2.1 will have the always-advance feature controllable by row.

Protecting against an abnormally large apply volume

Apply/MVS 1.2.1 customers should read the warning on page 22.

On some occasions, you may be concerned about the size of the change data answer set waiting to be applied. An Apply process that's been shut down for some time will attempt to 'catch up' at the next opportunity, possibly taking on a abnormally large workload in the process. You might consider adding a predicate similar to that shown in Figure 13, to limit the size of the change data answer set fetched, transmitted, temporarily filed and applied during any one subscription cycle. The example predicate in Figure 13 allows at most 1 hour's worth of change data to be applied at one time. Assuming a normal INTERVAL_MINUTES value of 20 and a two hour backlog of change data, this predicate will spread the backlog over three subscription cycles instead of just one.

```
[WHERE]
IBMSNAP_COMMITSEQ IN
(SELECT U2.IBMSNAP_COMMITSEQ FROM IBMSNAP_UOW U1, IBMSNAP_UOW U2,
pruning_cntl PC  /* you must look up the name of the pruning control table */
WHERE
U1.IBMSNAP_COMMITSEQ=PC.PRUNE_LIMIT_SEQ
AND U2.IBMSNAP_LOGMARKER<=U1.IBMSNAP_LOGMARKER + 1 HOUR)
```

Figure 13 Subqeuery placed in IBMSNAP_REF_CNTL.PREDICATES to restrict size of change data answer set.

When pruning doesn't seem to be effective

Sometimes, you'll see many rows remaining in the IBMSNAP_UOW table following pruning. Remember, if you have copies which only rarely receive updates, the CD table and the UOW table information will accumulate. As the progress of each copy is tracked by the last transaction *applied*, this situation may be aggravated if you have very restrictive filtering predicates for one or more subscriptions. In a manner of speaking, the UOW table can become constipated.

One technique to alleviate the problem is to regularly issue some dummy updates which will propagate to the otherwise infrequently updated copies. Then, their progress will be noticed during pruning and the CD and UOW tables trimmed down to a more manageable size.

New with Apply 1.2.1, you can use the ARCH_LEVEL field in the IBMSNAP_REF_CNTL table to mark those particular subscriptions which should have their LAST_DIFF_SEQUENCE and PRUNE_LIMIT_SEQ values always advanced, regardless of whether any change data rows satisfy the subscription predicates. Setting the ARCH_LEVEL column of the IBMSNAP_REF_CNTL table for the subscription row to ’0102’ will indicate use of the always-advance feature, on a subscription by subscription basis. Previous to Apply 1.2.1, only ARCH_LEVEL values of ’0101’ were valid.

Once must be careful when using the always-advance feature, due to the potential for data loss when using subscription customization techniques which reference LAST_DIFF_SEQUENCE and/or PRUNE_LIMIT_SEQ. In other words, ARCH_LEVEL=’0102’ should not be used if the referential integrity or large answer set techniques are in use for the subscription. DPropR/2 will never set ARCH_LEVEL to ’0102’, and will fail when opening a subscription with ARCH_LEVEL set to the ’0102’ level, issuing a ASN2204 message. The always-advance feature then, can only be accessed by direct SQL manipulation of the subscription definition.

For usage information, also refer to the "Referential integrity" section on page 30. Apply/MVS customers should also read "Concerns for customers with Apply/MVS 1.2.1" on page 22.

If you’re using a version of Apply prior to 1.2.1, or you cannot use ARCH_LEVEL=’0102’, you can *tickle* a slow subscription into reporting its progress by posting an update to a dummy row in the base table, a row which is also included in the copy. Following the propagation of the dummy row, the subscription’s progress in the pruning control table will advance and pruning will not be blocked.

Avoiding Capture/MVS cold starts

First off, Capture/MVS does not have perfect information during a warm start, particularly a warm start following an abnormal termination, such as an operator *cancel*. In such a scenario, the warm start table does not exist because Capture/MVS came down hard, as the result of the operator cancel. In this situation, Capture/MVS (Release 2 or later) will scan the UOW and CD tables, looking for an RBA that will give it a "last known" log position. If it can’t find something useful there (pruning may have emptied these tables), it will look in the CD_CNTL and pruning control tables for an RBA to use.

This sounds fine, but understand that Capture/MVS will only have information on the last update captured, not the last log record processed. If there has been lots of update activity, but none of it against tables which are being captured, there could be a sizeable window between the position in the log of the last captured update and the current log position. If you find that Capture/MVS is unable to warm start following an operator cancel, the cause is likely that the last known position has fallen outside of the active log range. When attempting the warm start, Capture/MVS will come up with what it thinks is a good re-start point, only to get an error return from DB2 when asking for a log record in that range. Problems of this type go away if you use DPropR 1.2.1 which include capture support for archive log access for DB2 MVS Version 4.

If you've still on DB2 MVS V3, I can think of two techniques to reduce the restart-window exposure:

1. increase the active log space allocation

2. create a small table with the change data capture attribute, register it as any other DPropR source table, and write a small application that commits an update to this table, say, every 5 minutes. That way, the warm start window will be limited to a maximum of 5 minutes worth of log data, as Capture/MVS will always be assured of seeing a fairly-current RBA in the UOW table.

Propagating to DB2/6000 Parallel Edition

Apply/6000 may be used to maintain copies on DB2/6000 PE, but there are a few things to consider, given that parallel edition has the normal restrictions associated with partitioned tables, i.e., the partitioning key cannot be *updated*. Therefore, some guidelines are necessary:

1. use a single-node nodegroup on DB2/6000 PE or some other server for storing control tables (the easiest way to use DPRTOOLS to define the subscriptions and designate, say, DB2 MVS as the control_server)

2. For point-in-time copies:

    a. If the user transactions update the partitioning key of the target table, define the target table on a single-node nodegroup.

b. If the user transactions do not update the partitioning key of the target table, you may spread the target table across a multi-node nodegroup. Be aware that the default partitioning key is only the first column of the primary key. If the primary key is made up of more than one column you may not get the results you were expecting. In general, you will want to define a nodegroup (not IBMDEFAULTGROUP) and an explicit partitioning key. The easiest way to gain control over the nodegroup selection is to use an alternative tool, such as DPRTOOLS, to define the subscriptions.

Aggregate copies and non-condensed CCD tables are not a problem, as they are only *appended*, not updated in place.

Another common problem in setting up propagation from, say DB2 on MVS to DB2/6000 PE is that both systems must be defined as *remote* severs to DDCS/6000. To date, there is very little experience to draw upon. You'll want to consider hiring professional service providers to asses the potential for using replication with your applications and database design.

When copying to Oracle and Sybase through DataJoiner

The IBM publication "Data Where You Need It, The DPropR Way!", IBM order number GG24-4492-00 contains examples of copying from DB2 sources into Oracle and Sybase databases, and *from* these sources into DB2 databases. Also included are sample triggers for capturing updates to Oracle tables. Still, there are a few details which are not covered in the book (affectionately known as a *red book* due to the color of the covers on this series of practical product guides).

The following text is taken from the DataJoiner README file:

```
5.0 Data Replication - Using DProp/R (Apply for AIX) with DataJoiner

  Apply for AIX can use DataJoiner to replicate data to and from Oracle
  or Sybase.  When using DataJoiner for replication,
  consider the following:

  5.1 As part of the install process for Apply, six links are performed
      to the shared directories. Two of these links reference database
      directories. These links must be modified to reference DataJoiner
      libraries rather than DB2 libraries as follows:

      ln -s /usr/lpp/asnapply/asnapply.bnd /usr/lpp/djx_01_01_0000/bnd
      ln -s /usr/lpp/asnapply/ASNLOAD.bnd /usr/lpp/djx_01_01_0000/bnd

  5.2 Set LIBPATH Before Using Apply for AIX with DataJoiner
```

Apply for AIX needs to know where to find the DataJoiner run-time libraries. You provide this to Apply/6000 by adding the directory path for the DataJoiner libraries to the LIBPATH environment variable used by the Apply/6000 user ID (login). Add the path:

```
INSTHOME/sqllib/lib
```

Where INSTHOME is the home directory of the DataJoiner instance you access from the Apply/6000 user ID.

When you set LIBPATH, we suggest you put the DataJoiner library directory path ahead of the /usr/lib and /lib directory paths. An example of setting LIBPATH from the Korn shell command line is:

```
export LIBPATH=/home/djinst1/sqllib/lib:$LIBPATH
```

Note: If /usr/lib and /lib are not listed in the current setting of LIBPATH, add them. They contain general-purpose shared libraries, such as the C run-time libraries. An example of setting LIBPATH with these is:

```
export LIBPATH=/home/djinst1/sqllib/lib:/usr/lib:/lib
```

5.3 DataJoiner Database Creation:

When replicating data from Oracle, ensure the DataJoiner database is created with 'COLLATE USING IDENTITY'.

5.4 Using Apply for AIX to Access DB2 Data Sources

When replicating data to or from any DB2 data source (ie. DB2/MVS, DB2/400, etc.), Apply should access the database directly. For this reason, accessing tables via DataJoiner nicknames is not appropriate.

To do this, catalog the DB2 database in the DataJoiner database and DCS directories (if a DRDA data source) as well as the node directory. Apply will utilize DataJoiner's DRDA Application Requestor function to connect to the database DRDA directly, outside DataJoiner's heterogeneous environment (i.e., without using PASSTHRU or NICKNAMEs).

Refer to 'AIX Distributed Database Connections Services/6000' SC09-1568 for more information on configuring this connection from DataJoiner to a DRDA Application Server.

5.5 Using Apply for AIX to Access non-IBM Data Sources

When replicating to/from non-IBM data sources, use the DataJoiner nickname function to access a table in Oracle or Sybase.

5.6 More Information

Refer to the ITSO red book: 'Data Where You Need It, The Dprop/R Way', GG24-4492.

5.7 DB2/2 v1 CSD Required When Propagating from DRDA

If you plan to have DataPropagator Relational/2 access DataJoiner through DB2/2 Version 1 and you are going to propagate data from a DRDA data source (such as DB2 for MVS), you must first install one of the following DB2/2 CSDs:

- For DB2/n 1 release 1, install DB2/2 CSD WR07030.
- For DB2/2 version 1 release 2, install DB2/2 CSD WR07035.

Without the fix, CONNECTs to remote DRDA databases through DataJoiner return the connect information in the wrong order. This causes DataPropagator to put the subscriber userid rather than the RDBname of the Data Server in the DATA_SERVER column of IBMSNAP_REF_CNTL.

5.8 Column type conversion

```
    When replicating to and from non-IBM databases to IBM databases
    it maybe necessary to change the COLTYPE and LENGTH values in
    SYSIBM.SYSCOLUMNSfor the Oracle or Sybase table nickname.
    For example to replicate
    DATE and TIME values to Oracle where only the DATE data type is
    supported, the following updates to SYSIBM.SYSCOLUMNS will cause
    the data to be replicated properly for the DATA0 and DATA1 columns
    in the BASE2ORA table in Oracle.

 db2 "update sysibm.syscolumns set coltype='DATE',length=4 where
      name='DATA0' and tbname='BASE2ORA'"

 db2 "update sysibm.syscolumns set coltype='TIME',length=3 where
      name='DATA1' and tbname='BASE2ORA'"

    To determine what changes to be made for a given table replication
     to Oracle or Sybase:
      a) Observe COLTYPE and LENGTH after the target table is created
         in the DataJoiner database.
         Note this target table would have been created by the DPropR/2
         subscription process.
         This target table is then dropped and the true target table
         in Oracle or Sybase is created.
      b) Compare these values with COLTYPE and LENGTH values created
         by the nickname command for
         the actual target table in Sybase and Oracle.
      c) If there are problems replicating data, consider updating
         SYSIBM.SYSCOLUMNS as illustrated
         in the above example.

    For replications from Oracle or Sybase to IBM, use a similar
    technique to allow DataJoiner to
    perform data transformations. Refer to the DataJoiner 'Application
    Programming and SQL Reference Supplement' for more
    information on data compatibility.
```

Copying to/from other DRDA Application Servers

It is possible to use Apply/2 and Apply/MVS to copy to or from, say DB2/VM or other DRDA server. Refer to appends in the DPROPR FORUM for reports of using Apply/MVS to copy from SQL/DS to DB2, and using Apply/2 to copy from SQL/DS to DB2/2, etc.

4/13/96 note: DB2 for AIX and DB2 for OS/2 Version 2 DRDA Application Servers are not compatible with DPropR Apply products prior to DPropR 1.2.1, due to an implementation restriction documented in the DB2 Version 2 Installation and Operations manual. These DRDA Servers do not handle CHAR FOR BIT DATA host variables properly, though the DPropR Apply logic in 1.2.1 successfully works around the server deficiencies.

Alternatively, you can use a common server Apply process to *push* updates to MVS or to an AS/400, but with loss of the block fetch performance optimization.

Running multiple apply programs

At some point, you may define so much data to be copied to a site that a single apply program instance cannot keep the copies as current as you would like. The answer is to run as many apply program instances as required to meet the demand. Of course, you may want to run additional apply programs just because you want to keep your subscriber authorizations as specific as possible.

Each apply program will use at most one SNA session at a time, so adding additional apply program executions will not quickly exhaust your available SNA session resources.

"As of" temporal querying or copying

```
SELECT IBMSNAP_LOGMARKER,
  HEX(IBMSNAP_INTENTSEQ),IBMSNAP_OPERATION,
  KEY1,DATA1
  FROM USRT012.CCD_TABLE
WHERE
USRT012.CCD_TABLE.IBMSNAP_OPERATION IN('I','U') AND
USRT012.CCD_TABLE.IBMSNAP_LOGMARKER =
  (SELECT MAX(X.IBMSNAP_LOGMARKER)
   FROM USRT012.CCD_TABLE X
   WHERE
   USRT012.CCD_TABLE.KEY1 = X.KEY1
   AND DATE(X.IBMSNAP_LOGMARKER) < '1993-01-03');
```

Figure 14 Temporal query: view as of end-of-day January 2, 1993.

Figure 14 illustrates a temporal query against a CCD table similar to the one shown in Figure 2, with the assumption that this CCD table represents a complete history. The result of the query is the same as if one had queried the user table back in January of 1993. The predicates shown could be included in the definition of a copy. Specifically, the CCD_TABLE referenced in Figure 14 is *complete*, *non-condensed* and *transaction consistent*.

Referential integrity

Apply/MVS 1.2.1 customers should read the warning on page 22.

You may wish to coordinate many subscriptions together, so that the set of copies will reflect unit of work consistency, with all copies updated to the same synch-point. Apply/x will only commit updates to one copy table at a time, so even through a set of subscriptions can be coordinated, there is an exposure that the set of copies will not reflect unit of work consistency while the copies are being updated. If the copies are updated during off-hours, as is common for decision support applications, temporary inconsistencies will go unnoticed.

The general scheme then is to pick a set of tables and choose one to be the *lead* or *parent* table. This may or may not be the *parent table* as far as any constraint is concerned, it is merely the table in the group which is updated most often. This table will be copied first each time, then other tables will be updated following, but will not progress past the synch-point reflected in the first copy. Once updates to *all* of the copies have been committed, separately, the set of copies will reflect unit of work consistency.

Figure 14, Figure 15 shows the predicate definition of a copy of a dependent table, including the subquery reference to the pruning control table associated with the parent table. The reference to IBMSNAP_COMMITSEQ is not qualified, as it might refer to either the column in the ASN.IBMSNAP_UOW table, or to the column of the same name in a CCD table associated with the base table, if present. The parent subscription must run at a higher priority than the dependent subscriptions, so that the apply of recent changes to the parent always precedes the apply of recent changes to the dependents.

[WHERE]	IBMSNAP_COMMITSEQ <= (SELECT PRUNE_LIMIT_SEQ FROM pruning_control WHERE pruning_control.COPY_OWNER = 'primary-owner' and pruning_control.COPY_TABLE = 'primary-table')

Figure 15 Dependent table copy referring to pruning control table for the parent change data table. Literals are in upper case.

The predicate reference to IBMSNAP_COMMITSEQ must be added *after* the copy is initialized. Otherwise, the apply program will look for this column in the user table and fail. An alternative would be to define a view over the user table that included a column IBMSNAP_COMMITSEQ which was a constant value of x'00000000000000000000'. This view must be registered in addition to the physical user table and the same change data table should be named in both IBMSNAP_CD_CNTL registration rows. There are *always* alternatives!

Normally, LAST_DIFF_SEQUENCE and PRUNE_LIMIT_SEQ only advance for a subscription as change data rows are applied. If no change data rows satisfy the subscription predicates, no rows are propagated to the copy and the state of the subscription does not advance. A new feature in DPropR 1.2.1 allows for advancing the subscription state, even for subscription cycles where no change data rows satisfy the predicates (see "Concerns for customers with Apply/MVS 1.2.1" on page 22 and "When pruning doesn't seem to be effective" on page 23).

New with Apply 1.2.1, you can use the ARCH_LEVEL field in the IBMSNAP_REF_CNTL table to mark those particular subscriptions which should have their LAST_DIFF_SEQUENCE and PRUNE_LIMIT_SEQ values always advanced, regardless of whether any change data rows satisfy the subscription predicates. Setting the ARCH_LEVEL column of the IBMSNAP_REF_CNTL table for the subscription row to '0102' will indicate use of the always-advance feature, on a subscription by subscription basis. Previous to Apply 1.2.1, only ARCH_LEVEL values of '0101' were valid.

Once must be careful when using the always-advance feature, due to the potential for data loss when using subscription customization techniques which reference LAST_DIFF_SEQUENCE and/or PRUNE_LIMIT_SEQ. In other words, ARCH_LEVEL='0102' should not be used if the referential integrity or large answer set techniques are in use for the subscription. DPropR/2 will never set ARCH_LEVEL to '0102', and will fail when opening a subscription with ARCH_LEVEL set to the '0102' level, issuing a ASN2204 message. The always-advance feature then, can only be accessed by direct SQL manipulation of the subscription definition.

In an "R.I." scenario, where one source table is designated as the lead or *parent* table, and other source tables are designated as *dependents*, at least within a subscription context, the greatest benefit is derived when using a mix of ARCH_LEVEL values when subscribing to these tables.

The subscription from the lead table would not contain any references to PRUNE_LIMIT_SEQ values in the predicates and would use ARCH_LEVEL = '0102' in order that the state of the lead subscription would always advance.

The subscriptions from the dependent tables would include references to the lead table's PRUNE_LIMIT_SEQ value, and use ARCH_LEVEL = '0101'. These subscriptions would only advance with respect to change data rows applied, so would not be exposed to a propagation data loss.

By using this combination of both subscription-state management capabilities, you'll gain benefits heretofore not possible. One problem with the "lead table" concept has been that, if no updates propagated to the copy of the lead table, then no updates would propagate to the copies of the dependent tables, either, as the dependent subscriptions would all be gated by the progress, however slow, of the lead subscription. If the lead table is updated only infrequently, propagation delays become unacceptably long.

Using ARCH_LEVEL='0102' to always advance the state of the lead table subscription, the subscriptions to the dependent tables are not blocked unnecessarily. Yet, the benefits of using the PRUNE_LIMIT_SEQ predicate references to prevent the dependent subscriptions from advancing *beyond* the state of the lead subscription are maintained. **Rather than being mutually exclusive, the two subscription-state management capabilities are nicely complimentary.**

Controlling the timing of copying

Apply/MVS 1.2.1 customers should read the warning on page 22.

It's possible to add predicates to your subscriptions which reference the time of day. For instance, you might include the expression:

HOUR(CURRENT TIMESTAMP) >= 17
AND HOUR(CURRENT TIMESTAMP) <= 20

Doing so would allow the update of your copy to be initiated between the hours of 5:00pm and 8:00pm. To make sure the update occurs, set the frequency often enough, such as every 90 minutes. Though a 90 minute refresh interval will cause the request to become active up to 16 times a day, it will only update the copy between 5:00 and 8:00pm. Especially if this is a local copy, there is little cost in momentarily activating a subscription definition that falls outside of its imposed timing *window*.

You can set the value of the LASTRUN column in the IBMSNAP_REF_CNTL table to adjust the daily subscription timing. Say that your subscription is currently running every 24 hours, each day about 10:00am. To adjust the timing to have the subscription run at 6:00pm each day, set the value of LASTRUN timestamp to 6:00pm of the previous day. That way, your Apply process will think that the next subscription cycle should begin at 6:00pm instead of 10:00am. From then on, the subscription will maintain (approximately) the 6:00pm daily schedule.

It's also possible to use a value of 0 for INTERVAL_MINUTES, to run your subscriptions *continuously*. You cannot set INTERVAL_MINUTES to 0 using DPropR/2 however, you must use SQL directly, or another administration tool, such as DPRTOOLS.

Loading large copies

Yes, you can perform your own load and fool the apply component into thinking it has initialized your large point in time copies. One possible scenario: Define a subscription but include a dummy predicate that restricts the copy to few or no rows. This is quite easy with a predicate such as saying CURRENT TIMESTAMP must be less than a 1980 date -- an impossible condition. The Apply component will update the pruning_cntl table and you will now have a synchronization timestamp. After creating the subscription, extract the table and load it using the REPLACE option. You may leave the IBMSNAP_LOGMARKER timestamp column null as this is a "full refresh".

Then, remove the dummy predicate from the subscription definition. Now, the Apply component will maintain the copy without having actually initialized it. The AUDIT_TRAIL statistics will show a full refresh of 0 rows. The Apply component will perform a differential refresh thereafter. As an extra measure, you will likely want to turn off the FULL_REFRESH_ENABLE flag in the CD_CNTL table at the data_server(source). This will prevent the Apply component from trying to initialize new copies via the normal on-line initialization process.

Here are the steps, in detail, for a DB2-to-DB2 subscription scenario:

A. start Capture/MVS, if it is not already running
B. create your subscription definition with:
   1. a dummy predicate in the *Where* box of the rows page to ensure a null (or small, it doesn't really matter) answer set
   2. set the frequency page *Iterations* value to receive copies 1 time only
C. allow the Apply/MVS program time to complete the "null" initialization of the copy -- you will see a row added to the IBMSNAP_TRAIL table noting the "initialization"
D. unload the data from the base table
   for DB2/2 copies, use the DB/2 EXPORT utility
E. load (replace) the data into the copy table
   for DB2/2 copies, use the DB/2 IMPORT utility
F. open the subscription and:
   1. remove the dummy predicate from the *Where* box of the rows page
   2. set the *Iterations* value on the frequency page to *Receive copies until Subscription is deactivated.*

Automating the large copy load

For copies on OS/2 and AIX, you can easily set up your Apply process to use a high speed load utility, running in the ASNLOAD exit. Sample exit implementations are available for both OS/2 and AIX which use the fast load utility that comes free with DB2 Common Server V2.

On MVS, there is no "callable" load utility for Apply/MVS to use. The following general procedures, which work on all platforms, will be most useful to MVS customers.

While the above procedure explained in the "Loading large copies" section works fine when loading one or two copies, you may wish to do a bit of programming and automate the loading of many DB2 MVS copies. To begin with, define all your subscriptions, but be sure to set *activate = no* on the frequency panel of the subscription notebook. The following steps should be run in a single MVS batch job, **after you've started Capture/MVS**, using appropriate condition code tests, etc.

A. Run an SQL application (DSNTEP2 for example) to update the pruning control table associated with the base table. Set *copy_server*, *copy_owner* and *copy_table* to the fully qualified name of your copy (*copy_server* can be any value, it is just a comment), *ibmsnap_logmarker* to a timestamp value approximating the current time (need not be accurate), *prune_limit_seq* to x'00000000000000000000', *cntl_server* to either the name of the data-server or copy-server -- wherever the IBMSNAP_REF_CNTL table is stored, and *copy_structure* matching the value of the same column in the IBMSNAP_REF_CNTL table. In the IBMSNAP_REF_CNTL definitions, set the value of *lastrun* to the same timestamp value set in the pruning control table row.

B. Run the DB2 REORG utility to unload the user table into a load format dataset.

C. Run the DB2 LOAD utility to load the unloaded user table data into your copy. Assuming the copy is a *point in time* structure, use a default constant for the value of the *ibmsnap_logmarker* column. This timestamp to be meaningful to those looking at the data should approximate the time of the unload.

D. Run an SQL application to set the *enable* column values for the subscriptions in the IBMSNAP_REF_CNTL table(s) to +1. This is the last step. As long as Capture/MVS and Apply/x are running, the copies should now only be differentially updated as the full refresh process has completed.

Defining size, compression, partitioning for DB2 CD or copy tables

DataPropagator Relational/2 will create your CD and copy tables automatically, in the DB2 database you specify, using space parameters derived from the source table, if such statistics are available. You may wish to define different space parameters, Version 3 compression, or other DDL specifics for your copy (or CD) table.

Begin by creating your subscription the normal way. Before closing the subscription notebook and committing your subscription definition, set the *activate* button to *no* on the frequency page of the notebook. Then, alter the tablespace with the desired characteristics. Recreate the copy table in the new tablespace and reorganize the tablespace. In DataPropagator Relational/2, open the subscription and set the *activate* button to *yes*. You can alternatively activate the subscription by setting the value of IBMSNAP_REF_CNTL.ENABLE for the subscription to 1.

Alternatively, you can set tablespace options explicitly using the DPRTOOLS, introduced in the "Tools" section on page 36.

Tools

Many of the advanced DPropR techniques involve use of registration and subscription definitions which cannot be created or maintained by the DPropR/2 user interface. In some cases, there are tools available to define & maintain these advanced registrations and subscriptions. You will find a few references still to the OS/2 REXX tools JOINREG, UNISUB and UNION, which used to be distributed with this document. All examples using these tools have been replaced with examples using the more powerful DPRTOOLS. The following sections will refer to tools available to IBMrs and customers through controlled distribution (DPRTOOLS, ASNBATCH).

DPRTOOLS

The DPRTOOLS are a collection of OS/2 REXX programs, accessed via a VX-REXX presentation manager interface. In order to allow you the flexibility of setting tablespace options when creating CD and copy tables, as well as giving you full control over the naming of tablespaces, tables and indexes, the registration and subscription functions of DPropR/2 are replaced. Once you begin using the flexibility the DPRTOOLS offer, you cannot go back to using DPropR/2, as it will be unable to maintain the registrations and subscriptions you've created.

The DPRTOOLS are not products, but a means for exploring new administrative functions in advance of full product development. Support is limited. Distribution is at no cost, but controlled. For details, IBMrs can refer to postings on the DPROPR FORUM on IBMPC.

For usage examples, see the "Using DPRTOOLS in a data warehouse view scenario" beginning on page 46 and "Using DPRTOOLS in a data warehouse union scenario" beginning on page 72.

DPRTOOLS global assumptions

1. DB2/400 is not supported by any of the tools.
2. Only 8-byte *alias* server names used. All data_server, cntl_server and copy_server names must fit into a flat 8-byte naming scheme and the same 8-byte names must uniquely identify a server from any vantage point.
3. DataJoiner is not explicitly supported.
4. Where wildcards are allowed, the input must either be a string with no '%' wildcards, or a pattern string with one or more '%' wildcards, with one a trailing '%' wildcard. '%' matches a substring of none or many arbitrary characters.
5. When a pattern string contains a '%' wildcard, any '_' characters also in the string will be interpreted as single-character wildcards.

TREG: table registration

1. Wildcards may be used to choose a range of CREATOR and NAME values from SYSIBM.SYSTABLES.
2. For MVS sources, registration assumes all columns of the source table.
3. For MVS sources, LONG VARCHAR columns represented by LONG VARCHAR columns in the CD table (DPropR/2 represents MVS LONG VARCHAR columns as VARCHAR columns in the CD table).
4. For DB2/CS, registration assumes all columns of types supported by Capture. BLOB, CLOB, DCLOB and LONG columns are not registered.
5. All column names must be upper case.
6. The 2-part names of the CD and pruning control tables may be set to any arbitrary values.
7. The names of the indexes for the CD and pruning control tables may be set to any arbitrary values.
8. The unique pruning control table index includes the COPY_SERVER column. DPropR/2 does not include this column when creating an index.
9. All manner of DDL options are supported. Any new options which may become available are supported via user editing of the REXX statements which generate the DB2 CREATE TABLESPACE statement.
10. Implicit tablespaces also supported, if the user prefers not to explicitly create tablespaces.
11. Capture of before-image columns only not supported.
12. Capture of after-image or both before-and-after column images is decided for the entire table, not on a column by column basis.
13. Names of CD and pruning control tables which match the BASE_OWNER and BASE_TABLE patterns are noted and ignored. It is not possible to register CD and pruning control tables.
14. It is not possible to register the ASN.IBMSNAP_UOW table. Attempts to do so will result in an error.
15. Checks, such as discovering which tablenames match the input pattern have already been registered, rely on the current values in ASN.IBMSNAP_CD_CNTL. Previously generated registrations must have been run, using RUNSQL or other means, in order for subsequent usage of TREG to have knowledge of these prior registrations.
16. View and alias names matching the BASE_OWNER and BASE_TABLE patterns are automatically excluded.
17. If the ASN.IBMSNAP_CD_CNTL table does not exist at the data_server, CREATE TABLE statements are automatically generated for ASN.IBMSNAP_CD_CNTL, ASN.IBMSNAP_CRITSEC, ASN.IBMSNAP_TRACE, and ASN.IBMSNAP_CCPPARMS. A single tablespace can be named as the default for these tables, or left unspecified to use implicit tablespaces.

18. The Capture component, and with it the IBMSNAP_TRACE and IBMSNAP_CCPPARMS tables, are installed separately.
19. All tables are registered for change capture, except for SYSIBM. catalog tables, which are registered without a CD table.
20. ALTER TABLE DATA CAPTURE CHANGES statements are automatically generated for tables being registered for change capture which do not have the CDC attribute. This is different from DPropR/2, which assumes it may not be appropriate to immediately ALTER a production table. Because the TREG registration is deferred, it is possible to run the ALTER TABLE statement, along with the rest of the registration statements, at an appropriate time.

VREG: view registration

1. No wildcards may be used, only one view may be registered at a time.
2. At least one of the view components must be a registered table.
3. Registered view components may or may not be registered for change capture.
4. Column names need not be unique among view components. Redundant column names may be qualified in the WHERE clause by 1-part or 2-part tablenames, or may be qualified with a correlation name.
5. The view name VIEW.VIEW may not be registered.
6. The maximum number of tables referenced in a view definition is 13.
7. Views over CCD tables are not supported.
8. One additional character is added to the end of the view name when creating additional views, so views being registered must have unqualified names 17 bytes (or less) in length.
9. For automatic unique index generation to work properly in subsequent subscription tasks, key columns should be selected when creating the view initially, in cases where there is a choice between selecting a column which is part of a unique index for one component, and an alternate column which is not part of any unique index.

10. The new "INNER", "OUTER" and "ON" keywords, introduced with DB2 MVS V4 to support outer join syntax, are not supported. Use of this syntax will result in the generation of incorrect CREATE VIEW statements, not detected until the statements are run.
11. If the application deletes matching rows in two or more of the tables referenced in a join view definition, rows which match according to one of the '=' predicates which define the join, then these deletes will not propagate. Applications which delete such matching rows require additional delete/delete view registrations, to be defined by the user, and additional subscriptions, also defined by the user, to ensure that deletes do propagate correctly.
12. View definitions must not contain subqueries.

CANCELR: cancel registrations

1. Wildcards may be used for BASE_OWNER and BASE_TABLE to choose among registrations to cancel.
2. A warning is issued when cancelling a registration which has dependent subscriptions, or which has dependent view registrations which have dependent subscriptions.
3. For view registrations to be properly cancelled, they must have been created using the JOINREG or VREG tools, which use the DESCRIPTION column of IBMSNAP_CD_CNTL to tie together the elements of a view registration.
4. Table and view registrations may be cancelled.
5. Tablespaces are dropped only if the last table in the tablespace is dropped as a result of the cancel task.

PROMOTER: promote registrations

1. Wildcards may be used for BASE_OWNER and BASE_TABLE to choose among registrations to promote.
2. All values are taken verbatim from the DPropR control tables and the DB2 catalog tables, with the exception of the names of implicit tablespaces. Implicit tablespaces are promoted as explicitly created tablespaces, with generated tablespace names.
3. DB2/CS tablespace definitions are not promoted, regardless.
4. DB2 MVS tablespaces are promoted, regardless of type (simple, segmented, partitioned).
5. For every promoted change data or pruning control table, all indexes are also promoted, regardless of type (simple, segmented, partitioned).

SUB1: subscribe to a registered table or view

1. No wildcards allowed.
2. Views registered by VREG may be referenced by the name of the view, and the subscription processed as a single task. DPropR/2 requires multiple subscription tasks to accomplish one logical view subscription.
3. Column subsetting only via view registration.
4. No support for base-aggregate or change-aggregate subscriptions.
5. SQL-before, SQL-after specification not directly supported. SQL-after statement generated as comment in output, for ease of specifying SQL-after statement during editing before running subscription definition.
6. No support for column renaming, other than through view registrations.
7. The 2-part names of the CD and pruning control tables may be set to any arbitrary values.
8. The names of the indexes for the CD and pruning control tables may be set to any arbitrary values.
9. The unique pruning control table index includes the COPY_SERVER column. DPropR/2 does not include this column when creating an index.
10. All manner of DDL options are supported. Any new options which may become available are supported via user editing of the REXX statements which generate the DB2 CREATE TABLESPACE statement.
11. Implicit tablespaces also supported, if the user prefers not to explicitly create tablespaces.
12. Explicit starting time for the subscription may be set in the defaults.
13. If the IBMSNAP_REF_CNTL table does not exist at the cntl_server, CREATE TABLE statements are automatically generated for IBMSNAP_REF_CNTL, IBMSNAP_REF_COLS and IBMSNAP_TRAIL tables. A single tablespace can be named as the default for these tables, or left unspecified to use implicit tablespaces.
14. If the IBMSNAP_ROUTING table does not exist at the cntl_server, a CREATE TABLE statement is automatically generated for IBMSNAP_ROUTING
15. Only one cntl_server may be used for an Apply process, and the IBMSNAP_ROUTING table is also stored at this server. The Apply process must connect to the control server initially. This is different than the DPropR/2 convention, which assumes the Apply process initially connects to the local_server, where it will find the IBMSNAP_ROUTING table, which will direct it to one *or more* cntl_servers.
16. A CCD subscription may not be defined referencing a view registration which includes more than one change data source. There is a test to detect this condition. To understand the restriction, assume the existence of a subscription which uses the CCD copy as a *source*, in addition to the multiple subscriptions which would be needed to maintain the CCD from the multi-part view registration. Following the first apply into the CCD, the IBMSNAP_COMMITSEQ values in the table would advance. If the subscription using the CCD as a source were to run next, it would assume that all transactions up to the highest IBMSNAP_COMMITSEQ value had been applied to the CCD table. Assume that only the first of many of the view subscriptions has run. Change data from those remaining view subscriptions having IBMSNAP_COMMITSEQ values below the highest value already posted in the CCD will forever be ignored by the subscription reading from the CCD.

17. CCD subscription assumptions are the same as DPropR/2: if the CCD is local and there is yet no 'internal' CCD defined, then the copy will be a non-complete CCD. In all other cases the copy will be complete. The use can choose between condensed and non-condensed.
18. If the copy table is found to exist, the names of the columns in the table will be checked against all of the columns in the subscription. If there are columns which are in the subscription but missing from the copy table, including any IBMSNAP* columns, SUB1 will end in error, giving a message stating the missing column name.

SUBMANY: subscribe to many registered tables

1. Wildcards may be used to choose among BASE_OWNER, BASE_TABLE values in ASN.IBMSNAP_CD_CNTL.
2. The SUB1 tool is called repeatedly, so any SUB1 restrictions other than the use of wildcards, also applies to SUBMANY.
3. View registrations are ignored. This is because of the complexity of trying to couple wildcard processing with the transparency mechanism for DPropR V1 view registrations, relying on ASN.IBMSNAP_CD_CNTL.DESCRIPTION values.
4. All copies are assumed condensed and the unique index is generated by default, containing all columns which are contained in any unique index on the source table. Subscriptions to source tables which have no such unique index columns will fail.

CANCELS: cancel subscriptions

1. Wildcards may be used for COPY_SERVER, COPY_OWNER and COPY_TABLE to choose among subscriptions to cancel.
2. Table and view subscriptions may be cancelled.
3. Tablespaces are dropped only if the last table in the tablespace is dropped as a result of the cancel task.

PROMOTES: promote subscriptions

1. Wildcards may be used for COPY_SERVER, COPY_OWNER and COPY_TABLE to choose among subscriptions to promote.
2. All values are taken verbatim from the DPropR control tables and the DB2 catalog tables, with the exception of the names of implicit tablespaces. Implicit tablespaces are promoted as explicitly created tablespaces, with generated tablespace names.
3. DB2/CS tablespace definitions are not promoted, regardless.
4. DB2 MVS tablespaces are promoted, regardless of type (simple, segmented, partitioned).
5. For every promoted copy table or pruning control table, all indexes are also promoted, regardless of type (simple, segmented, partitioned).

REPORTS: report on subscriptions

1. Wildcards are not supported.
2. The report will attempt to list all subscriptions dependent on any table or view registration for a given data_server.
3. Of the status indicators:
   (1) 'failed' is IBMSNAP_REF_CNTL.STATUS= -1
   (2) 'disabled' is IBMSNAP_REF_CNTL.STATUS= 0
   (3) 'not run' is IBMSNAP_REF_CNTL.LASTRUN is null
   (4) 'late' is IBMSNAP_REF_CNTL.LASTRUN < CURRENT TIMESTAMP - (INTERVAL_MINUTES + late_threshold MINUTES)
   (5) 'on time' otherwise
4. If any connections are down, the report generation will stop, in error.

RUNSQL: run an SQL file

1. Run a file with SQL statements, as generated by one of the other DPRTOOLS.
2. Statements other than CONNECT statements will be processed using separate PREPARE and INSERT statements.
3. Commit points are noted, so that if there is a failure midpoint in running the file, the position of the last commit statement processed in the file is displayed. This is useful when wanting to later re-run only the statements not yet committed.

ASNBATCH

A REXX command line program, also in limited distribution, which refines optional DPropR/2 session trace files for reuse. The SQL statements from the DPropR/2 session, once stored in a file, can then be easily edited before reusing the statements in another context, such as cloning subscriptions for a second copy server. Overall, this tool is somewhat equivalent to the *promote* functions of DPRTOOLS.

Background on Replicating Joins

Unlike most replication products, DPropR Apply can maintain join copies *incrementally*. Before jumping into a discussion of specific examples, let's cover some background information so that you'll understand why join replication is so *difficult*. First off, let's cover a few intuitive approaches to join replication that *don't work* in practice.

It isn't enough just to join recent changes to one table with recent changes to another table. Say that TAB1 contains a record of bank customers' automatic teller transactions. Say that TAB2 contains each customer's name, address, telephone number and bank branch number. Assume that a customer will typically change their banking branch or home address, etc., only once every two years. If you wanted to incrementally maintain a copy of the join of TAB1 with TAB2, it would be nice if this could be done without any more information than the lists of recent updates to TAB1 and TAB2. Given the low frequency of updates to TAB2, the average propagation delay of joining a changed row in TAB1 with a counterpart changed row in TAB2 would be something over a *year*, with some updates taking *decades* to propagate.

Joins can't generally be computed at the target site. You're out of luck trying to compute the join at the target site, unless one of the following restrictions holds:

1. All data from the source site is replicated at the target site.

2. All foreign key values can be accurately predicted and all relevant data permanently maintained at the target site (potential savings over #1 above, if possible).

   There is always the chance that an update to TAB1 will change a foreign key column that will cause the row in TAB1 to be joined with a different row in TAB2 -- a row which may, or may not be present in the copy of TAB2.

3. Join column data, foreign keys and primary keys, are never updated.

In addition, you must be able to route each update to the appropriate site or sites before the join is computed (an example to follow will illustrate why this might be more difficult than you would think).

Even when computing the join at the source site, there are potential complexities that you must watch out for, and handle appropriately. One would quite naturally use SQL predicates, referring to a source table, to describe the *fragment* of the source table replicated at the target site. What if a row that used to satisfy the SQL predicates associated with a replicated fragment is updated so that it no longer satisfies said predicates? Well, it may be that the row now satisfies some other set of predicates and should appear in a different replicated fragment somewhere else. Regardless, the old row, which is still in the original replicated fragment, must be removed. Update operations can be difficult to replicate correctly. We'll dub this the "fragment migration problem". You'll see this problem dealt with in the "Database Partitioning" example just ahead.

Delete operations can also be difficult to replicate correctly. We've already discussed why directly joining together change-data from two or more tables is problematic. You might then think that the contrasting approach -- taking a join of each table's change-data joined with all the other base tables, then unioned with the next set of change-data joined with the other base tables, etc., would yield promising results. Promising, yes, but not without problems when related rows are deleted from multiple tables in the same timeframe (unit of replication, be it an individual transaction, or an interval which would include multiple transactions). You see, if the change-data row for TAB1 indicates a delete, and the related row or rows in TAB2 have also been deleted, then there is no way to propagate the deletes to *either* TAB1 or TAB2 when relying solely on the technique of joining each set of change-data, separately, with the base tables.

As the above paragraphs show, you must have knowledge of the updating *applications*, not just the data, in order to replicate a join satisfactorily.

1. Are the copy's primary key columns ever *updated*?
2. Are the columns referenced in the fragmentation predicates *updated*?
3. Are related rows in different tables ever *deleted* within the same transaction?

A 'yes' to any of the above three questions signals the need for additional complexity for your join replication definitions. Neither JOINREG nor the VREG function in the DPRTOOLS handle the three problems listed above for you. You must customize your subscriptions to handle these problem scenarios, should they pertain to your application.

Getting down to the specifics of using DPropR for join replication, let's accept the SQL 'CREATE VIEW' statement as a practical method for describing a join. Each entry in the FROM clause we'll call a *component* of the join. Here is a list of rules regarding join components:

1. If a join component is static, is a reference table only, you need not register this table for change capture. In fact, you need not register the table, period.

2. If the component is subject to insert, update or delete operations, you must register the component for change capture.

    a. If a column is going to be part of the primary key of the copy table, and is subject to being updated in place, then you must register both the *before* and *after* image of the column for change capture.

b. If a column is going to be referenced in the subscription predicates, and is subject to being updated in place, then you must register both the *before* and *after* image of the column for change capture.

The join examples to follow rely on the technique of defining a view of one component's change data table joined with the other component tables. We'll rely on the additional technique of using an SQL-after delete statement in the subscription definition to handle both the fragment migration problem and the multi-table delete problem, should either potential problem arise.

Even taking all of the above details into consideration, there are still a few potential surprises to take into account:

1. If you modify the subscription predicates in the future, you'll need to concern yourself with removing rows in the copies that should no longer be there if they wouldn't satisfy the redefined subscription predicates, and as well, you'll need to revisit the question of deciding for which columns you need to capture both *before* and *after* images.

2. If the behavior of the applications change, you may need to revisit the rules for replicating *updates* and *deletes*.

3. Keep in mind that change data cannot be staged in a CCD table as a join if more than one of the components of the join, union, or join-union is registered for change capture.

    a. When joining two or more change data rows, which values are you going to use for the single set of IBMSNAP_INTENTSEQ, IBMSNAP_COMMITSEQ values in the target CCD?

    b. If the two change data streams being unioned were coming from different source databases, the log sequence numbers, the IBMSNAP_INTENTSEQ and IBMSNAP_COMMITSEQ values, would be utterly incompatible.

c. Even if the two change data streams came from a single source database, because they will be applied to the CCD staging table in separate transactions, there is the potential for a race condition between the two (or more) feeder subscriptions and any other subscription which reads from the CCD. As the Apply process reading from the CCD to maintain a downstream copy will keep track of its progress by remembering a high IBMSNAP_COMMITSEQ value in LAST_DIFF_SEQUENCE, it would miss rows that might appear at a later time in the CCD staging table, if those rows had IBMSNAP_COMMITSEQ values lower than that already posted in LAST_DIFF_SEQUENCE.

The DPropR/2 administration component will not allow the registration of views, only tables. Also, do not use the DPropR/2 CANCEL action against a view registration, as the pruning control table which is shared with the physical registration, and possibly with other view registrations, will be dropped. The cancel functions included in the DPRTOOLS will handle view registrations created using either JOINREG or the VREG function of the DPRTOOLS, and also protect against dropping a non-empty tablespace.

The DPRTOOLS allow you to easily register your join-views over previously registered components (or static tables, which may or may not be registered). Once the views are registered, you'll find it easiest to continue using DPRTOOLS to define and maintain your view subscriptions.

Using DPRTOOLS in a data warehouse view scenario

note: this example has not been tested

In this section, we'll use the tables in Figure 22 on page 60 that are used extensively in the "Database Partitioning" example.

Figure 16 shows the opening DPRTOOLS window. From here, we'll select the *Register tables* action, assuming that the tables in Figure 22 have not yet been registered.

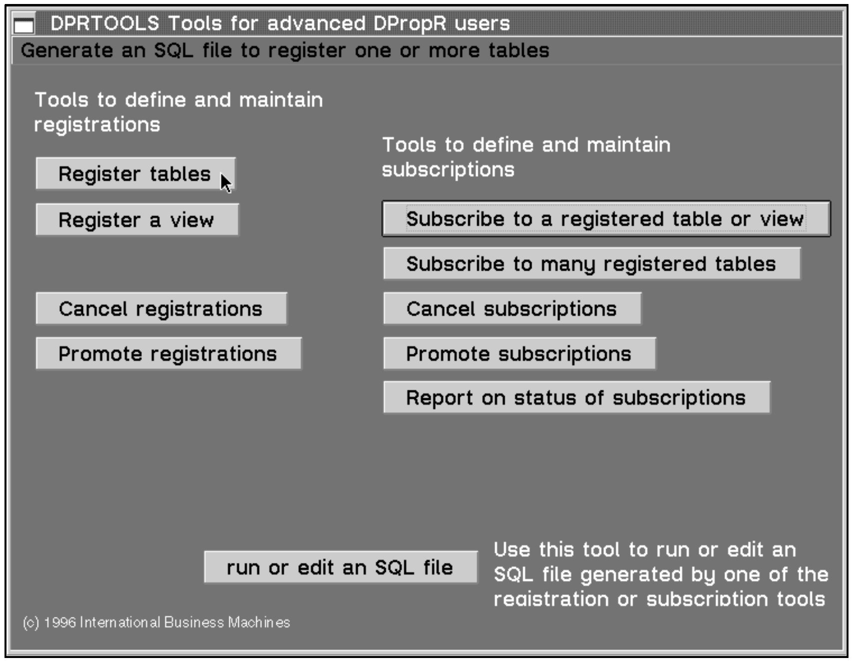

Figure 16 The DPRTOOLS main window.

Figure 17 on page 47 shows the table registration window. Notice that there is no column list -- all columns will be registered. Both the base table qualifier and base table name input fields accept a '%' wildcard, which will allow us to register all four tables from Figure 22 with this single registration action.

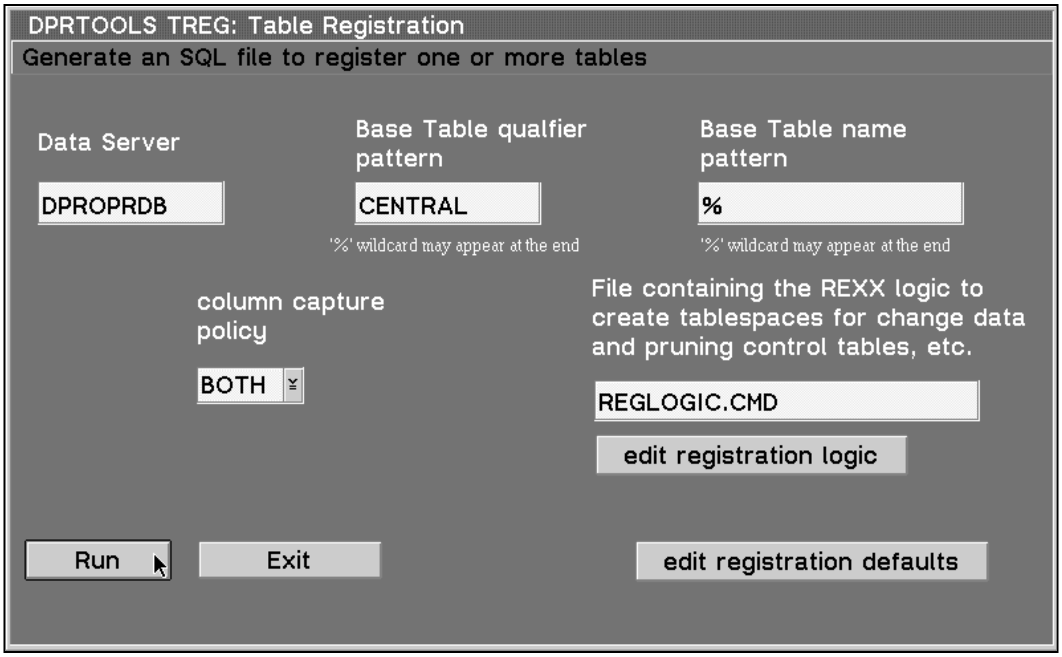

Figure 17 The DPRTOOLS table registration window.

After filling in the entry fields as shown in Figure 17 and clicking on the *Run* pushbutton, a pop-up window appears asking if the file should be merely displayed to the console window or both displayed and saved in a file. Once the choice has been made to save the generated output, a standard dialog window appears to prompt for the name of the output file. In this case, we'll store the generated output in a file *TREG.SQL*, the contents of which follows:

```
--* echo input: TREG DPROPRDB CENTRAL % BOTH REGLOGIC.CMD
--*
-- using REGLOGIC.CMD as the  REXX logic filename
-- TREG test version 02/27/96
-- USERID=USERID   DATA_SERVER alias=DPROPRDB 11 Apr 1996 8:23pm
-- using REXX defaults file TREGDFLT.CMD

-- If you don't see: '-- now done interpreting...' then check your REXX code
-- now done interpreting REXX logic file TREGDFLT.CMD
-- connect to the data-server
CONNECT TO DPROPRDB;
-- 4 candidate registrations, but 13 already known to be registered

-- The following tables are candidates for registration:
--   1 table          CENTRAL.ACCOUNT
--   2 table          CENTRAL.BALANCE
--   3 table          CENTRAL.BRANCH
--   4 table          CENTRAL.CUSTOMER

-- registration candidate #1 CENTRAL.ACCOUNT
-- using REXX logic file REGLOGIC.CMD

-- If you don't see: '-- now done interpreting...' then check your REXX code
```

```
-- in REGLOGIC.CMD, about to create a change data tablespace
CREATE TABLESPACE TS232609 MANAGED BY DATABASE
 USING (FILE 'C:\TEST\TS232609.F1' 500);

-- now done interpreting REXX logic file REGLOGIC.CMD
-- selecting 'X' as the before-image prefix character
-- create the change data table for CENTRAL.ACCOUNT
CREATE TABLE CENTRAL.CD0000000000000001(IBMSNAP_UOWID CHAR(10) FOR BIT
DATA NOT NULL,IBMSNAP_INTENTSEQ CHAR(10) FOR BIT DATA NOT NULL,
IBMSNAP_OPERATION CHAR(1) NOT NULL,CUSTOMER_ID INTEGER NOT NULL,
XCUSTOMER_ID INTEGER,ACCT_ID INTEGER NOT NULL,XACCT_ID INTEGER) IN
TS232609;

-- create the index for the change data table for CENTRAL.ACCOUNT
CREATE INDEX CENTRAL.CDI000000000000001 ON CENTRAL.CD0000000000000001(
IBMSNAP_UOWID ASC);

-- create the pruning control table for CENTRAL.ACCOUNT
CREATE TABLE CENTRAL.PC0000000000000001(COPY_SERVER CHAR(18) NOT NULL,
COPY_OWNER CHAR(18) NOT NULL,COPY_TABLE CHAR(18) NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,PRUNE_LIMIT_SEQ CHAR(10) FOR BIT DATA,
CNTL_SERVER CHAR(18) NOT NULL,COPY_STRUCTURE SMALLINT NOT NULL) DATA
CAPTURE CHANGES IN TS232609;

-- create the unique index for the pruning control table
CREATE UNIQUE INDEX CENTRAL.PCI000000000000001 ON
CENTRAL.PC0000000000000001(COPY_SERVER,COPY_OWNER,COPY_TABLE);

-- insert a registration record into ASN.IBMSNAP_CD_CNTL
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CCD_OWNER,CCD_TABLE,CCD_OLDEST_CHANGE,CD_CONDENSED,CD_CONSISTENT,
CD_COMPLETE,CCD_CONDENSED,CCD_CONSISTENT,CCD_COMPLETE,BASE_STRUCTURE,
BASE_CONDENSED,BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,
DESCRIPTION,PREFIX_CHAR) VALUES( 'CENTRAL','ACCOUNT','ASN',
'IBMSNAP_CRITSEC','CENTRAL','CD0000000000000001','CENTRAL',
'PC0000000000000001',NULL,1,NULL,NULL,NULL,'N','T','N',NULL,NULL,NULL,
1,'Y','T','Y','USERID  ','0101',NULL,'X');

-- registration candidate #2 CENTRAL.BALANCE
-- using REXX logic file REGLOGIC.CMD

-- If you don't see: '-- now done interpreting...' then check your REXX code
-- in REGLOGIC.CMD, about to create a change data tablespace
CREATE TABLESPACE TS232715 MANAGED BY DATABASE
 USING (FILE 'C:\TEST\TS232715.F1' 500);

-- now done interpreting REXX logic file REGLOGIC.CMD
-- selecting 'X' as the before-image prefix character
-- create the change data table for CENTRAL.BALANCE
CREATE TABLE CENTRAL.CD0000000000000002(IBMSNAP_UOWID CHAR(10) FOR BIT
DATA NOT NULL,IBMSNAP_INTENTSEQ CHAR(10) FOR BIT DATA NOT NULL,
IBMSNAP_OPERATION CHAR(1) NOT NULL,ACCT_ID INTEGER NOT NULL,XACCT_ID
INTEGER,ACCT_BALANCE INTEGER,XACCT_BALANCE INTEGER) IN TS232715;

-- create the index for the change data table for CENTRAL.BALANCE
CREATE INDEX CENTRAL.CDI000000000000002 ON CENTRAL.CD0000000000000002(
IBMSNAP_UOWID ASC);

-- create the pruning control table for CENTRAL.BALANCE
CREATE TABLE CENTRAL.PC0000000000000002(COPY_SERVER CHAR(18) NOT NULL,
```

```
COPY_OWNER CHAR(18) NOT NULL,COPY_TABLE CHAR(18) NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,PRUNE_LIMIT_SEQ CHAR(10) FOR BIT DATA,
CNTL_SERVER CHAR(18) NOT NULL,COPY_STRUCTURE SMALLINT NOT NULL) DATA
CAPTURE CHANGES IN TS232715;

-- create the unique index for the pruning control table
CREATE UNIQUE INDEX CENTRAL.PCI000000000000002 ON
CENTRAL.PC0000000000000002(COPY_SERVER,COPY_OWNER,COPY_TABLE);

-- insert a registration record into ASN.IBMSNAP_CD_CNTL
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CCD_OWNER,CCD_TABLE,CCD_OLDEST_CHANGE,CD_CONDENSED,CD_CONSISTENT,
CD_COMPLETE,CCD_CONDENSED,CCD_CONSISTENT,CCD_COMPLETE,BASE_STRUCTURE,
BASE_CONDENSED,BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,
DESCRIPTION,PREFIX_CHAR) VALUES( 'CENTRAL','BALANCE','ASN',
'IBMSNAP_CRITSEC','CENTRAL','CD0000000000000002','CENTRAL',
'PC0000000000000002',NULL,1,NULL,NULL,NULL,'N','T','N',NULL,NULL,NULL,
1,'Y','T','Y','USERID  ','0101',NULL,'X');

-- registration candidate #3 CENTRAL.BRANCH
-- using REXX logic file REGLOGIC.CMD

-- If you don't see: '-- now done interpreting...' then check your REXX code
-- in REGLOGIC.CMD, about to create a change data tablespace
CREATE TABLESPACE TS232800 MANAGED BY DATABASE
 USING (FILE 'C:\TEST\TS232800.F1' 500);

-- now done interpreting REXX logic file REGLOGIC.CMD
-- selecting 'X' as the before-image prefix character
-- create the change data table for CENTRAL.BRANCH
CREATE TABLE CENTRAL.CD0000000000000003(IBMSNAP_UOWID CHAR(10) FOR BIT
DATA NOT NULL,IBMSNAP_INTENTSEQ CHAR(10) FOR BIT DATA NOT NULL,
IBMSNAP_OPERATION CHAR(1) NOT NULL,CUSTOMER_ID INTEGER NOT NULL,
XCUSTOMER_ID INTEGER,BRANCH_NUMBER INTEGER,XBRANCH_NUMBER INTEGER) IN
TS232800;

-- create the index for the change data table for CENTRAL.BRANCH
CREATE INDEX CENTRAL.CDI000000000000003 ON CENTRAL.CD0000000000000003(
IBMSNAP_UOWID ASC);

-- create the pruning control table for CENTRAL.BRANCH
CREATE TABLE CENTRAL.PC0000000000000003(COPY_SERVER CHAR(18) NOT NULL,
COPY_OWNER CHAR(18) NOT NULL,COPY_TABLE CHAR(18) NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,PRUNE_LIMIT_SEQ CHAR(10) FOR BIT DATA,
CNTL_SERVER CHAR(18) NOT NULL,COPY_STRUCTURE SMALLINT NOT NULL) DATA
CAPTURE CHANGES IN TS232800;

-- create the unique index for the pruning control table
CREATE UNIQUE INDEX CENTRAL.PCI000000000000003 ON
CENTRAL.PC0000000000000003(COPY_SERVER,COPY_OWNER,COPY_TABLE);

-- insert a registration record into ASN.IBMSNAP_CD_CNTL
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CCD_OWNER,CCD_TABLE,CCD_OLDEST_CHANGE,CD_CONDENSED,CD_CONSISTENT,
CD_COMPLETE,CCD_CONDENSED,CCD_CONSISTENT,CCD_COMPLETE,BASE_STRUCTURE,
BASE_CONDENSED,BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,
DESCRIPTION,PREFIX_CHAR) VALUES( 'CENTRAL','BRANCH','ASN',
'IBMSNAP_CRITSEC','CENTRAL','CD0000000000000003','CENTRAL',
```

```
'PC0000000000000003',NULL,1,NULL,NULL,NULL,'N','T','N',NULL,NULL,NULL,
1,'Y','T','Y','USERID  ','0101',NULL,'X');

-- registration candidate #4 CENTRAL.CUSTOMER
-- using REXX logic file REGLOGIC.CMD

-- If you don't see: '-- now done interpreting...' then check your REXX code
-- in REGLOGIC.CMD, about to create a change data tablespace
CREATE TABLESPACE TS232887 MANAGED BY DATABASE
 USING (FILE 'C:\TEST\TS232887.F1' 500);

-- now done interpreting REXX logic file REGLOGIC.CMD
-- selecting 'X' as the before-image prefix character
-- create the change data table for CENTRAL.CUSTOMER
CREATE TABLE CENTRAL.CD0000000000000004(IBMSNAP_UOWID CHAR(10) FOR BIT
DATA NOT NULL,IBMSNAP_INTENTSEQ CHAR(10) FOR BIT DATA NOT NULL,
IBMSNAP_OPERATION CHAR(1) NOT NULL,CUSTOMER_ID INTEGER NOT NULL,
XCUSTOMER_ID INTEGER,NAME CHAR(18),XNAME CHAR(18),ADDRESS CHAR(20),
XADDRESS CHAR(20)) IN TS232887;

-- create the index for the change data table for CENTRAL.CUSTOMER
CREATE INDEX CENTRAL.CDI000000000000004 ON CENTRAL.CD0000000000000004(
IBMSNAP_UOWID ASC);

-- create the pruning control table for CENTRAL.CUSTOMER
CREATE TABLE CENTRAL.PC0000000000000004(COPY_SERVER CHAR(18) NOT NULL,
COPY_OWNER CHAR(18) NOT NULL,COPY_TABLE CHAR(18) NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,PRUNE_LIMIT_SEQ CHAR(10) FOR BIT DATA,
CNTL_SERVER CHAR(18) NOT NULL,COPY_STRUCTURE SMALLINT NOT NULL) DATA
CAPTURE CHANGES IN TS232887;

-- create the unique index for the pruning control table
CREATE UNIQUE INDEX CENTRAL.PCI000000000000004 ON
CENTRAL.PC0000000000000004(COPY_SERVER,COPY_OWNER,COPY_TABLE);

-- insert a registration record into ASN.IBMSNAP_CD_CNTL
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CCD_OWNER,CCD_TABLE,CCD_OLDEST_CHANGE,CD_CONDENSED,CD_CONSISTENT,
CD_COMPLETE,CCD_CONDENSED,CCD_CONSISTENT,CCD_COMPLETE,BASE_STRUCTURE,
BASE_CONDENSED,BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,
DESCRIPTION,PREFIX_CHAR) VALUES( 'CENTRAL','CUSTOMER','ASN',
'IBMSNAP_CRITSEC','CENTRAL','CD0000000000000004','CENTRAL',
'PC0000000000000004',NULL,1,NULL,NULL,NULL,'N','T','N',NULL,NULL,NULL,
1,'Y','T','Y','USERID  ','0101',NULL,'X');

COMMIT;

-- Satisfactory completion at 8:23pm
```

As you can see above, the DDL, including tablespace options have been nicely generated. To customize the tablespace and index definitions, it's possible to either:

1. edit the REXX logic prior to generating the registrations
2. choose an alternate REXX logic file, using different REXX files, say, for large tables or small tables
3. edit the generated output before running the SQL

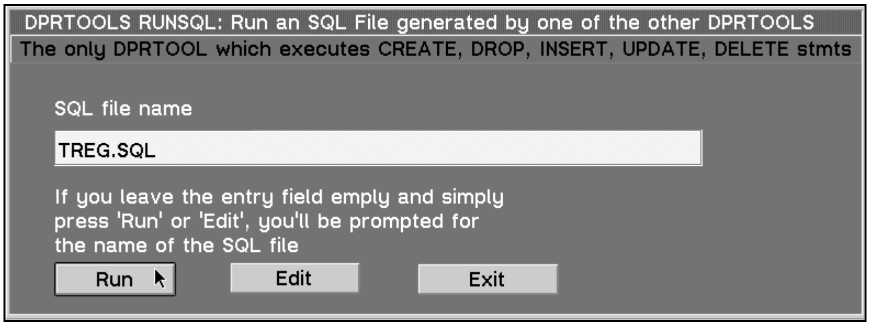

Figure 18 The DPRTOOLS run or edit SQL file window.

At this point, the SQL statements comprising the four registrations have been generated, but not yet run. By choosing the *run or edit an SQL file* option from the main window, the window shown in Figure 18 appears, prompting for the filename. Clicking on the *Run* pushbutton now will run and commit the registration SQL statements.

Now that all four table are registered, let's proceed to defining a join view over all four of the bank application tables. From the OS2 command line, enter:

```
C:\DB2 CREATE VIEW WAREHOUS.VIEWJOIN4 AS SELECT
A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C,
CENTRAL.BALANCE D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND
A.CUSTOMER_ID=C.CUSTOMER_ID AND C.ACCT_ID=D.ACCT_ID
```

Once the view is created, we can use the *Register a view* option from the main window to bring up the view registration window in Figure 19. Clicking on the *Run* pushbutton causes a similar series of events as with the table registration process. First, an SQL file is generated, then it must be run to complete the view registration. The contents of the file with the generated view definitions is included below.

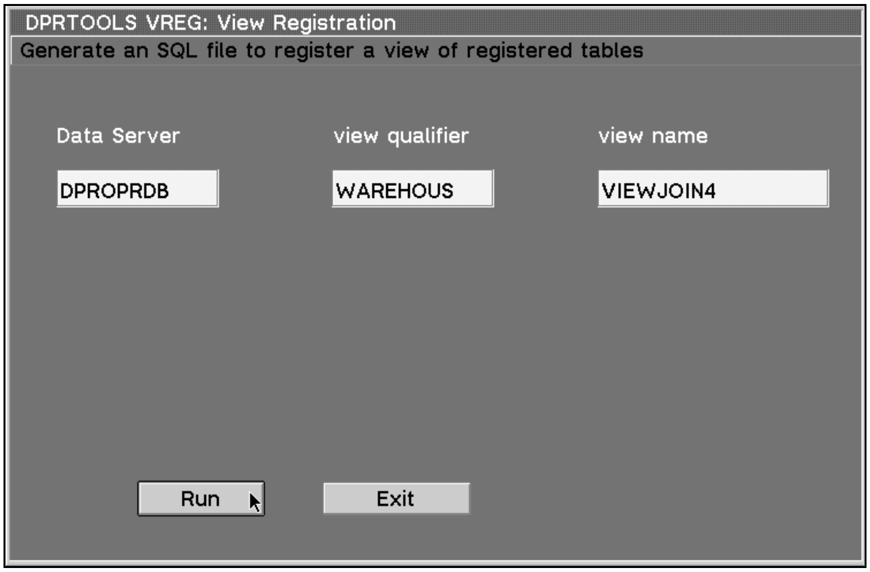

Figure 19 The DPRTOOLS view registration window.

```
-- VREG test version 02/27/96
-- input view OWNER=WAREHOUS  input view NAME=VIEWJOIN4
-- USERID=USERID   DATA_SERVER alias=DPROPRDB 11 Apr 1996 8:31pm
-- connect to the data server
CONNECT TO DPROPRDB;

-- register the input view
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CD_CONDENSED,CD_CONSISTENT,CD_COMPLETE,BASE_STRUCTURE,BASE_CONDENSED,
BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,DESCRIPTION,
PREFIX_CHAR)VALUES('WAREHOUS','VIEWJOIN4','ASN','IBMSNAP_CRITSEC',NULL,
NULL,'CENTRAL','PC0000000000000002',NULL,1,NULL,NULL,NULL,1,'Y','T',
'Y','USERID  ','0101','MODEL VIEW REGISTRATION FOR WAREHOUS.VIEWJOIN4',
'X');

-- create a new base view for component 1
CREATE VIEW WAREHOUS.VIEWJOIN4A AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- create the change data view for component 1
CREATE VIEW WAREHOUS.VIEWJOIN4B AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM CENTRAL.BRANCH A,CENTRAL.CUSTOMER
B,CENTRAL.ACCOUNT C, CENTRAL.CD0000000000000002 D WHERE
A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;
```

```
-- register the base and change data views for component 1
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CD_CONDENSED,CD_CONSISTENT,CD_COMPLETE,BASE_STRUCTURE,BASE_CONDENSED,
BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,DESCRIPTION,
PREFIX_CHAR)VALUES(’WAREHOUS’,’VIEWJOIN4A’,’ASN’,’IBMSNAP_CRITSEC’,
’WAREHOUS’,’VIEWJOIN4B’,’CENTRAL’,’PC0000000000000002’,
X’00000000000000000001’,1,’N’,’T’,’N’,1,’Y’,’T’,’Y’,’USERID  ’,’0101’,
’VIEW REGISTRATION FOR WAREHOUS.VIEWJOIN4’,’X’);

-- create a new base view for component 2
CREATE VIEW WAREHOUS.VIEWJOIN4C AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- create the change data view for component 2
CREATE VIEW WAREHOUS.VIEWJOIN4D AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM CENTRAL.CD0000000000000003 A,
CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE D WHERE
A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;

-- register the base and change data views for component 2
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CD_CONDENSED,CD_CONSISTENT,CD_COMPLETE,BASE_STRUCTURE,BASE_CONDENSED,
BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,DESCRIPTION,
PREFIX_CHAR)VALUES(’WAREHOUS’,’VIEWJOIN4C’,’ASN’,’IBMSNAP_CRITSEC’,
’WAREHOUS’,’VIEWJOIN4D’,’CENTRAL’,’PC0000000000000003’,
X’00000000000000000001’,1,’N’,’T’,’N’,1,’Y’,’T’,’Y’,’USERID  ’,’0101’,
’VIEW REGISTRATION FOR WAREHOUS.VIEWJOIN4’,’X’);

-- create a new base view for component 3
CREATE VIEW WAREHOUS.VIEWJOIN4E AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- create the change data view for component 3
CREATE VIEW WAREHOUS.VIEWJOIN4F AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM CENTRAL.BRANCH A,CENTRAL.CUSTOMER
B,CENTRAL.CD0000000000000001 C, CENTRAL.BALANCE D WHERE
A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;

-- register the base and change data views for component 3
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CD_CONDENSED,CD_CONSISTENT,CD_COMPLETE,BASE_STRUCTURE,BASE_CONDENSED,
BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,DESCRIPTION,
PREFIX_CHAR)VALUES(’WAREHOUS’,’VIEWJOIN4E’,’ASN’,’IBMSNAP_CRITSEC’,
’WAREHOUS’,’VIEWJOIN4F’,’CENTRAL’,’PC0000000000000001’,
X’00000000000000000001’,1,’N’,’T’,’N’,1,’Y’,’T’,’Y’,’USERID  ’,’0101’,
```

```
'VIEW REGISTRATION FOR WAREHOUS.VIEWJOIN4','X');

-- create a new base view for component 4
CREATE VIEW WAREHOUS.VIEWJOIN4G AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- create the change data view for component 4
CREATE VIEW WAREHOUS.VIEWJOIN4H AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE FROM CENTRAL.BRANCH A,
CENTRAL.CD0000000000000004 B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE D
WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;

-- register the base and change data views for component 4
INSERT INTO ASN.IBMSNAP_CD_CNTL(BASE_OWNER,BASE_TABLE,
CRIT_SECTION_OWNER,CRIT_SECTION_TABLE,CD_OWNER,CD_TABLE,
PRUNE_CNTL_OWNER,PRUNE_CNTL_TABLE,CD_OLDEST_CHANGE,FULLREFRESHENABLE,
CD_CONDENSED,CD_CONSISTENT,CD_COMPLETE,BASE_STRUCTURE,BASE_CONDENSED,
BASE_CONSISTENT,BASE_COMPLETE,REGISTRAR,ARCH_LEVEL,DESCRIPTION,
PREFIX_CHAR)VALUES('WAREHOUS','VIEWJOIN4G','ASN','IBMSNAP_CRITSEC',
'WAREHOUS','VIEWJOIN4H','CENTRAL','PC0000000000000004',
X'00000000000000000001',1,'N','T','N',1,'Y','T','Y','USERID  ','0101',
'VIEW REGISTRATION FOR WAREHOUS.VIEWJOIN4','X');

COMMIT;
-- Satisfactory completion at 8:31pm
```

Let's assume we've run the view registration statements above and move on to *subscribing* to the join view in Figure 20.

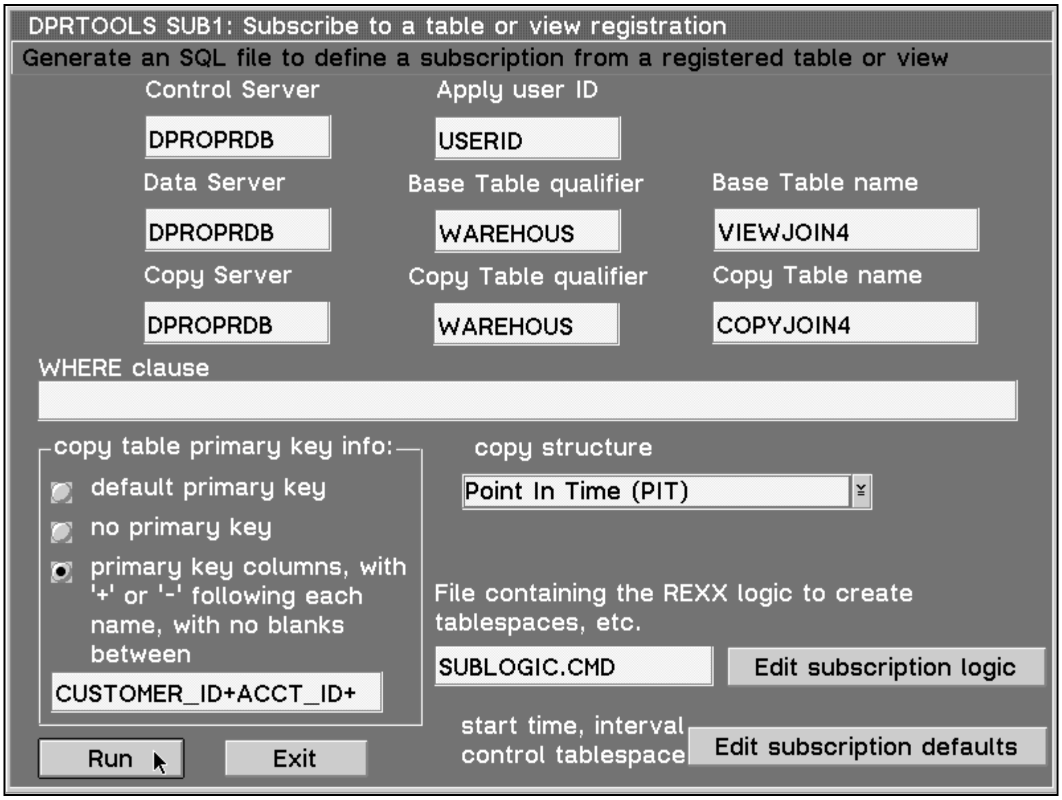

Figure 20 The DPRTOOLS single-subscription window.

Quite different from the subscription notebook of DPropR/2, the DPRTOOLS subscription window is just that, a *single window*. Many of the details in the DPropR/2 subscription notebook are hidden away in the two REXX files accessible via the *Edit subscription logic* and *Edit subscription defaults* push buttons. Choosing to supply the copy table primary key column list saves scanning SYSIBM.SYSKEYS and SYSIBM.SYSINDEXES, which can be a big time saver when subscribing from a DB2 MVS source database with a large catalog. The generated subscription definition is shown below:

```
--* echo input: SUB1 DPROPRDB USERID DPROPRDB WAREHOUS VIEWJOIN4
--* DPROPRDB WAREHOUS COPYJOIN4 CUSTOMER_ID+ACCT_ID+ PIT SUBLOGIC.CMD
--*
-- SUB1 test version 04/05/96
-- using REXX defaults file SUB1DFLT.CMD

--* If you don't see: '-- now done interpreting...' then check your
--* REXX code
--*
-- now done interpreting REXX logic file SUB1DFLT.CMD

--* connect to the data_server
--*
CONNECT TO DPROPRDB;
```

```
-- current USERID=USERID   DATA_SERVER alias=DPROPRDB 11 Apr 1996 8:35pm
-- using SUBLOGIC.CMD as the  REXX logic filename
--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000002(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('DPROPRDB','WAREHOUS',
'COPYJOIN4','DPROPRDB',4);

--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000003(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('DPROPRDB','WAREHOUS',
'COPYJOIN4','DPROPRDB',4);

--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000001(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('DPROPRDB','WAREHOUS',
'COPYJOIN4','DPROPRDB',4);

--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000004(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('DPROPRDB','WAREHOUS',
'COPYJOIN4','DPROPRDB',4);

-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1 ,'DPROPRDB','WAREHOUS','VIEWJOIN4A',100,10,'WAREHOUS',
'COPYJOIN4',NULL,'AO',0,NULL,NULL,12,'Y','Y','T',4,'0101','DPROPRDB',
NULL,NULL,NULL);

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','CUSTOMER_ID','Y',1,
'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','BRANCH_NUMBER','N',2,
'BRANCH_NUMBER');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','NAME','N',3,'NAME');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','ADDRESS','N',4,'ADDRESS');
```

```
-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','ACCT_ID','Y',5,'ACCT_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','ACCT_BALANCE','N',6,
'ACCT_BALANCE');

-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1 ,'DPROPRDB','WAREHOUS','VIEWJOIN4C',101,10,'WAREHOUS',
'COPYJOIN4',NULL,'AO',0,NULL,NULL,12,'Y','Y','T',4,'0101','DPROPRDB',
NULL,NULL,NULL);

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','CUSTOMER_ID','Y',1,
'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','BRANCH_NUMBER','N',2,
'BRANCH_NUMBER');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','NAME','N',3,'NAME');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','ADDRESS','N',4,'ADDRESS');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','ACCT_ID','Y',5,'ACCT_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','ACCT_BALANCE','N',6,
'ACCT_BALANCE');
```

```
-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1 ,'DPROPRDB','WAREHOUS','VIEWJOIN4E',102,10,'WAREHOUS',
'COPYJOIN4',NULL,'AO',0,NULL,NULL,12,'Y','Y','T',4,'0101','DPROPRDB',
NULL,NULL,NULL);

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','CUSTOMER_ID','Y',1,
'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','BRANCH_NUMBER','N',2,
'BRANCH_NUMBER');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','NAME','N',3,'NAME');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','ADDRESS','N',4,'ADDRESS');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','ACCT_ID','Y',5,'ACCT_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','ACCT_BALANCE','N',6,
'ACCT_BALANCE');

-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1 ,'DPROPRDB','WAREHOUS','VIEWJOIN4G',103,10,'WAREHOUS',
'COPYJOIN4',NULL,'AO',0,NULL,NULL,12,'Y','Y','T',4,'0101','DPROPRDB',
NULL,NULL,NULL);

-- edit template for an SQL-after statement to run at the COPY_SERVER
--* INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
--* BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
--* LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
--* INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
--* COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
--* VALUES (1 ,'DPROPRDB',' ',' ',200,10,'WAREHOUS','COPYJOIN4',NULL,
--* 'EI',0,NULL,NULL,12,' ',' ',' ',0,'0101','DPROPRDB',NULL,NULL,
--* '<your SQL statement here>');
```

```
--*
-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','CUSTOMER_ID','Y',1,
'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','BRANCH_NUMBER','N',2,
'BRANCH_NUMBER');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','NAME','N',3,'NAME');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','ADDRESS','N',4,'ADDRESS');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','ACCT_ID','Y',5,'ACCT_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','ACCT_BALANCE','N',6,
'ACCT_BALANCE');

-- the copy table does not yet exist
-- using REXX logic file SUBLOGIC.CMD

--* If you don't see: '-- now done interpreting...' then check your
--* REXX code
--*
-- in SUBLOGIC.CMD, about to create a copy table tablespace
CREATE TABLESPACE TS355609 MANAGED BY DATABASE
 USING (FILE 'C:\TEST\TS355609.F1' 500);

-- now done interpreting REXX logic file SUBLOGIC.CMD

-- create the copy table WAREHOUS.COPYJOIN4
CREATE TABLE WAREHOUS.COPYJOIN4(IBMSNAP_LOGMARKER TIMESTAMP,
CUSTOMER_ID INTEGER NOT NULL,BRANCH_NUMBER INTEGER,NAME  CHAR(18),
ADDRESS  CHAR(20),ACCT_ID INTEGER NOT NULL,ACCT_BALANCE INTEGER) IN
TS355609;

-- create an index for the copy WAREHOUS.COPYJOIN4
CREATE UNIQUE INDEX WAREHOUS.COPYJOIN4 ON WAREHOUS.COPYJOIN4(
CUSTOMER_ID ASC,ACCT_ID ASC);

--*
--* commit work at DPROPRDB
--*
COMMIT;

-- Satisfactory completion of SUB1 at 8:35pm
```

Notice that all four subscriptions were generated in a single, logical action. The qualifier of the copy table, WAREHOUS, was chosen *arbitrarily*. The DPropR Apply process running under the USERID user ID will maintain the copy table WAREHOUSE.COPYJOIN4 as an inner join of CENTRAL.BRANCH, CENTRAL.CUSTOMER, CENTRAL.ACCOUNT and CENTRAL.BALANCE tables.

Target site joins

When you're willing and able to copy all of the related data you'll ever need to the target site, an alternative to using view registration is to simply define point in time (which are *complete*) subscriptions for all the join components, then define a join-view at the target site over all the copies (see Figure 21).

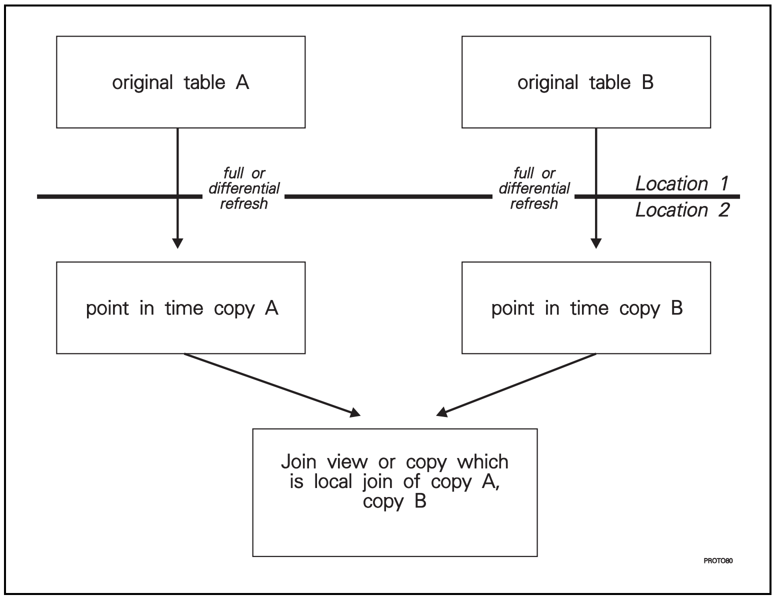

Figure 21 Copy components of a join, then access via a join view.

The join will be recalculated each time you reference the view. Expensive, maybe, but simple.
Still, the integrity of this view is affected by the degree of inconsistency you allow between the copies input into the join.

Database Partitioning

May 11, 1996 Note: this example has not been tested.

If you are trying to copy data from a central database to geographically disbursed branch offices, you may find that many of your production tables do not have any geographic information in them. Typically, production databases are *normalized*, and the geographic information, your desired *database* *partitioning* key, is only present in a related table. Consider the production tables in Figure 22 on page 60, where only the BRANCH table contains the BRANCH_NUMBER column. What you would like to do is copy data from all the other tables to their appropriate branches, that is, based on BRANCH_NUMBER value.

```
CREATE TABLE CENTRAL.BRANCH (
CUSTOMER_ID INT NOT NULL,
BRANCH_NUMBER INT)
DATA CAPTURE CHANGES

CREATE TABLE CENTRAL.CUSTOMER(
CUSTOMER_ID INT NOT NULL,
NAME CHAR(18),
ADDRESS CHAR(20))
DATA CAPTURE CHANGES

CREATE TABLE CENTRAL.ACCOUNT(
CUSTOMER_ID INT NOT NULL,
ACCT_ID INT NOT NULL)
DATA CAPTURE CHANGES

CREATE TABLE CENTRAL.BALANCE(
ACCT_ID INT NOT NULL,
ACCT_BALANCE INT )
DATA CAPTURE CHANGES
```

Figure 22 A fictitious production banking database, with BRANCH_NUMBER the database partitioning key.

This is most often more complicated than it sounds. First, notice that the BRANCH_ID column is not present in any of the other tables (CUSTOMER, ACCOUNT, BALANCE). Therefore, it is not possible to simply define a subscription with a filtering predicate referencing BRANCH_ID. To do that, you would first have to change the database schema to include the BRANCH_ID column in all tables, and modify your production transactions to maintain these redundant BRANCH_ID columns. Interestingly, this technique is commonly used by customers using competing replication products which lack DPropR's ability to propagate through *views*.

If the BRANCH table were completely static, never updated, then the subscriptions to the CUSTOMER, ACCOUNT and BALANCE tables could simply reference the BRANCH table in a subquery within the where clause of the subscription (see Figure 23), avoiding the need for view registrations and multiple subscriptions per copy table. We'll assume though that the BRANCH table is not static.

This further complicates the "database partitioning problem" for a normalized database such as shown in Figure 22, as the logical migration of rows in the CUSTOMER, ACCOUNT and BALANCE tables might be driven off of a delete *from* and insert *into* the BRANCH table, or merely an update to an existing customer's entry in the BRANCH table. In these cases, there are no captured updates to either the CUSTOMER, ACCOUNT or BALANCE tables, yet a series of deletes must take place against the copies of these tables in the customer's former branch office database, and a series of inserts must take place against the copies of these tables in the customer's new branch office database, when an update to CENTRAL.BRANCH.BRANCH_NUMBER occurs. To accomplish this expansion, derived solely by changes captured from the CENTRAL.BRANCH table, join views are necessary which will cause virtual change data rows for the CUSTOMER, ACCOUNT and BALANCE tables to be generated for captured updates to the BRANCH table. This scheme then, accommodates applications which perform exacting customer migrations, and those which may only modify the BRANCH table, either by deleting the old branch record and inserting the new branch record, or by simply updating the BRANCH record in place (the most difficult case to replicate correctly).

```
CENTRAL.CUSTOMER (CUSTOMER_ID, NAME, ADDRESS)

predicate:
[WHERE] CUSTOMER_ID IN
(SELECT B.CUSTOMER_ID FROM CENTRAL.BRANCH B
WHERE B.BRANCH_NUMBER=12)
```

Figure 23 When the BRANCH table is static, single subscriptions with subqueries referencing the BRANCH table may be used in place of join registrations and multiple view subscriptions.

In this detailed example, we'll show the views and predicates necessary to route the change data correctly, and also show the DELETE statements necessary to clean up each of the tables when a customer has moved on to another branch. To fully support the migration of all related records when a BRANCH_NUMBER value is updated, join registrations must be used, and multiple subscriptions each to maintain the copies of the CUSTOMER, ACCOUNT and BALANCE tables.

The subscription predicates will refer to BRANCH_NUMBER, so, to enable the scheme, both the before-image and after-image of BRANCH_NUMBER need to be captured. Assume that the before-image column prefix is 'X'. The before-image column XBRANCH_NUMBER need *not* be included in each of the copies of the BRANCH table, though this column may be included, as shown in Figure 26 on page 64. The columns CUSTOMER_ID and ACCT_ID are copy table key columns, which makes them candidates for before-image capture, but we'll assume that applications do not update these values in place, so that the before-image of these columns need not be captured.

For each of the tables BRANCH, CUSTOMER, ACCOUNT and BALANCE in Figure 22, assume that each table has been created or altered for change data capture, that each is registered for update propagation. No subscriptions will reference the registrations for the CUSTOMER, ACCOUNT and BALANCE tables. Instead, views will be created and registered for these three tables and all subscriptions to these tables will in fact reference the view registrations. Only the registration of the BRANCH table will be directly referenced in a subscription.

Registering the views

```
CREATE VIEW CENTRAL.CUSTOMER_VIEW AS SELECT
A.CUSTOMER_ID,NAME,ADDRESS,BRANCH_NUMBER FROM
CENTRAL.CUSTOMER A,CENTRAL.BRANCH B
WHERE A.CUSTOMER_ID=B.CUSTOMER_ID

CREATE VIEW CENTRAL.ACCOUNT_VIEW AS SELECT
A.CUSTOMER_ID,ACCT_ID,BRANCH_NUMBER FROM
CENTRAL.ACCOUNT A,CENTRAL.BRANCH B
WHERE A.CUSTOMER_ID=B.CUSTOMER_ID

CREATE VIEW CENTRAL.BALANCE_VIEW AS SELECT
A.ACCT_ID,ACCT_BALANCE,BRANCH_NUMBER FROM
CENTRAL.BALANCE A,CENTRAL.ACCOUNT B,CENTRAL.BRANCH C
WHERE A.ACCT_ID=B.ACCT_ID AND
B.CUSTOMER_ID=C.CUSTOMER_ID
```

Figure 24 Views defining joins of the CUSTOMER, ACCOUNT and BALANCE tables, separately, with the BRANCH table.

First, create the three views shown in Figure 24, then use the view registration function of DPRTOOLS to individually register each view (see "Using DPRTOOLS in a data warehouse view scenario" on page 46 for a step-by-step view registration example).

```
BASE_OWNER         BASE_TABLE         CD_TABLE           PRUNE_CNTL_TABLE
------------------ ------------------ ------------------ ------------------
CENTRAL            ACCOUNT            CD0000000000000001 PC0000000000000001
CENTRAL            ACCOUNT_VIEW       -                  PC0000000000000003
CENTRAL            ACCOUNT_VIEWA      ACCOUNT_VIEWB      PC0000000000000003
CENTRAL            ACCOUNT_VIEWC      ACCOUNT_VIEWD      PC0000000000000001
CENTRAL            BALANCE            CD0000000000000002 PC0000000000000002
CENTRAL            BALANCE_VIEW       -                  PC0000000000000003
CENTRAL            BALANCE_VIEWA      BALANCE_VIEWB      PC0000000000000003
CENTRAL            BALANCE_VIEWC      BALANCE_VIEWD      PC0000000000000001
CENTRAL            BALANCE_VIEWE      BALANCE_VIEWF      PC0000000000000002
CENTRAL            BRANCH             CD0000000000000003 PC0000000000000003
CENTRAL            CUSTOMER           CD0000000000000004 PC0000000000000004
CENTRAL            CUSTOMER_VIEW      -                  PC0000000000000003
CENTRAL            CUSTOMER_VIEWA     CUSTOMER_VIEWB     PC0000000000000003
CENTRAL            CUSTOMER_VIEWC     CUSTOMER_VIEWD     PC0000000000000004

  14 record(s) selected.
```

Figure 25 Contents (abbreviated) of ASN.IBMSNAP_CD_CNTL following the registration of ACCOUNT_VIEW, BALANCE_VIEW and CUSTOMER_VIEW.

Figure 25 shows the result of a query against ASN.IBMSNAP_CD_CNTL following the running of the SQL files with the view registrations. Don't be intimidated by all the additional entries.

Creating the subscriptions

```
BRANCH12.BRANCH (CUSTOMER_ID, BRANCH_NUMBER,
XBRANCH_NUMBER, IBMSNAP_OPERATION,
IBMSNAP_INTENTSEQ, IBMSNAP_COMMITSEQ,
IBMSNAP_LOGMARKER)

copy_complete='y', copy_condensed='y', copy_structure=CCD

predicate:
[WHERE] BRANCH_NUMBER=12 OR XBRANCH_NUMBER=12
```

Figure 26 The table description and subscription predicate for the copy of the BRANCH table.

Define a condensed (*one change per key*) CCD staging table subscription for the copy of the BRANCH table, as noted in Figure 26. This will ensure that even deleted rows appear at the remote site, and also provides an active notification of customers who have migrated out of branch 12 (when XBRANCH_NUMBER=12 but BRANCH_NUMBER does not). This active notification of migrations is the basis for the delete statements in the SQL-after processing of the other subscriptions. Via this notification mechanism, and the use of the delete statements during SQL-after processing, a customer migration caused by an update to the BRANCH_NUMBER column in the CENTRAL.BRANCH table will result in all related records for that customer being deleted from the copies of the CUSTOMER, ACCOUNT and BALANCE tables in the database at the customer's former bank branch. The use of a delete statement during SQL-after processing also, conveniently, handles multi-table delete situations, so that even if deletes to not propagate directly to the CUSTOMER, ACCOUNT and BALANCE copies, a delete propagating to the BRANCH copy is sufficient to cause the related rows to be deleted from the CUSTOMER, ACCOUNT and BALANCE copies.

The CCD structure is necessary because the operation column IBMSNAP_OPERATION is needed to positively distinguish customers with closed accounts, indicated by a delete to the BRANCH table, from customers who are migrating into this branch and who's entry in the BRANCH table merely has not propagated yet to this site. There are three interesting cases to keep in mind, with respect to delete processing:

1. When a delete to the BRANCH table is interpreted as the closing of all the customer's accounts, requiring deletion of all related rows in copies of the CUSTOMER, ACCOUNT and BALANCE tables.

2. When an out-migration is indicated by an update to the BRANCH table, requiring deletion of all related rows in copies of the CUSTOMER, ACCOUNT and BALANCE tables.

3. When an in-migration is indicated by either an insert or update to the BRANCH table, requiring retention of all related rows in copies of the CUSTOMER, ACCOUNT and BALANCE tables. In the in-migration scenario, rows may appear in the copies of the CUSTOMER, ACCOUNT and BALANCE tables *before* the appearance of the related row in the copy of the BRANCH table, so the mere *absence* of a row in the BRANCH table should not signal deletions of related rows in the other tables.

If your application is coded such that when a customer is dropped, one or more rows are deleted from the CUSTOMER, ACCOUNT and BALANCE tables and at the same time the customer's row is deleted from the BRANCH table, then the deletions to the CUSTOMER, ACCOUNT and BALANCE tables will not propagate directly, and not occur until the after the deletion to the BRANCH table is propagated, and only then by the SQL-after delete statements. This is because the view registrations based on CUSTOMER_VIEW, ACCOUNT_VIEW and BALANCE_VIEW will block the propagation of deletes. Take for example a delete to the CUSTOMER table -- when there is no longer any row with matching CUSTOMER_ID to join with in the BRANCH table, rows in change data table cannot propagate. The

SQL-after delete statement included in the subscription will finally delete the row from the copy of the CUSTOMER table, but only after the subscription for BRANCH12.BRANCH propagates a row indicating either:

1. a changed value of BRANCH_NUMBER for the particular customer, meaning an out-migration to another branch

2. the IBMSNAP_OPERATION field indicates a delete, representing a closing of this customer's accounts.

Creating the view subscriptions

Use DPRTOOLS to subscribe to the three registered views. Figure Figure 27 shows the DPRTOOLS panel settings used to create the subscription to CUSTOMER_VIEW, with the generated output following the figure.

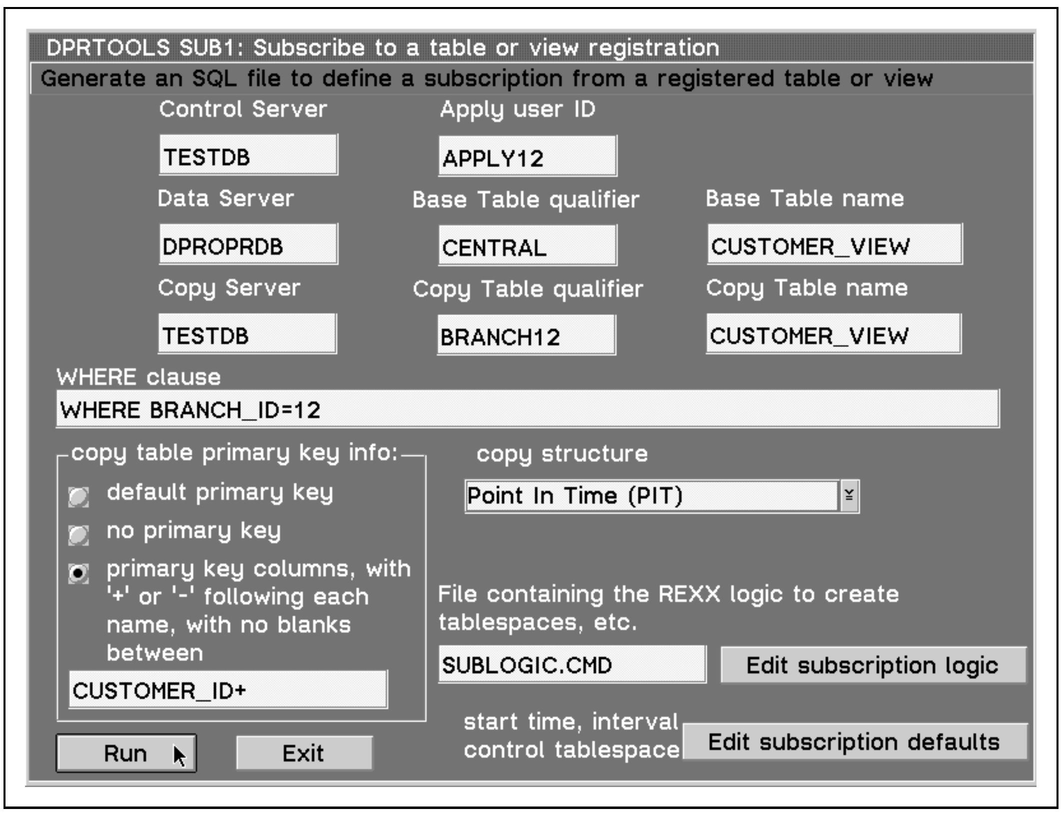

Figure 27 Subscribing to CENTRAL.CUSTOMER_VIEW.

This is the output generated from the operation in Figure 27:

```
--* echo input: SUB1 TESTDB APPLY12 DPROPRDB CENTRAL CUSTOMER_VIEW
--* TESTDB BRANCH12 CUSTOMER_VIEW CUSTOMER_ID+ PIT SUBLOGIC.CMD WHERE
--* BRANCH_ID=12
-- SUB1 test version 05/08/96
-- using REXX defaults file SUB1DFLT.CMD

--* If you don't see: '-- now done interpreting...' then check your
--* REXX code
--*
-- now done interpreting REXX logic file SUB1DFLT.CMD

--* connect to the data_server
--*
CONNECT TO DPROPRDB;

-- current USERID=USERID   DATA_SERVER alias=DPROPRDB 10 May 1996 6:04pm
-- using SUBLOGIC.CMD as the  REXX logic filename
--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000003(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('TESTDB','BRANCH12',
'CUSTOMER_VIEW','TESTDB', 4);

--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000004(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('TESTDB','BRANCH12',
'CUSTOMER_VIEW','TESTDB', 4);

-- commit work at server DPROPRDB
COMMIT;
-- switch context to server
CONNECT TO TESTDB;
-- subscriber 'APPLY12' does not yet have subscription control tables defined
-- create the IBMSNAP_REF_CNTL table and index
CREATE TABLE APPLY12.IBMSNAP_REF_CNTL (ENABLE SMALLINT NOT NULL,
DATA_SERVER CHAR(18) NOT NULL,BASE_OWNER CHAR (18) NOT NULL,BASE_TABLE
CHAR(18) NOT NULL,STMT_NUMBER SMALLINT NOT NULL,PRIORITY SMALLINT NOT
NULL,COPY_OWNER CHAR (18) NOT NULL,COPY_TABLE CHAR(18) NOT NULL,
LAST_DIFF_SEQUENCE CHAR(10) FOR BIT DATA,RUNMODE CHAR(2) NOT NULL,
STATUS SMALLINT NOT NULL,IBMSNAP_LOGMARKER TIMESTAMP,LASTRUN TIMESTAMP,
INTERVAL_MINUTES INT NOT NULL,COPY_CONDENSED CHAR(1) NOT NULL,
COPY_COMPLETE CHAR(1) NOT NULL,COPY_CONSISTENT CHAR(1) NOT NULL,
COPY_STRUCTURE SMALLINT NOT NULL,ARCH_LEVEL CHAR(4) NOT NULL,
COPY_SERVER CHAR(18) NOT NULL,EXPIRES DATE,PREDICATES VARCHAR (512),
SQL_STMT VARCHAR(1024)) IN DPROPRTSPACE;

CREATE UNIQUE INDEX APPLY12.XREF_CNTL ON APPLY12.IBMSNAP_REF_CNTL (
COPY_SERVER,COPY_OWNER,COPY_TABLE,STMT_NUMBER);

-- insert the IBMSNAP_REF_CNTL global control row
INSERT INTO APPLY12.IBMSNAP_REF_CNTL (ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,RUNMODE,STATUS,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER) VALUES(1 , '' , '' , '' , 0 , 0
, '' , '' , '' , 0 , 1 , '' , '' , '' , 0 , '0101' , '');

-- create the IBMSNAP_REF_COLS table and index
CREATE TABLE APPLY12.IBMSNAP_REF_COLS(COPY_SERVER CHAR(18) NOT NULL,
```

```
COPY_OWNER CHAR(18) NOT NULL,COPY_TABLE CHAR(18) NOT NULL,STMT_NUMBER
SMALLINT NOT NULL,COL_TYPE CHAR(1) NOT NULL,TARGET_NAME CHAR(18) NOT
NULL,IS_KEY CHAR(1) NOT NULL,COLNO SMALLINT NOT NULL,EXPRESSION
VARCHAR(254) NOT NULL) IN DPROPRTSPACE;

CREATE INDEX APPLY12.XREF_COLS ON APPLY12.IBMSNAP_REF_COLS (
COPY_SERVER,COPY_OWNER,COPY_TABLE,STMT_NUMBER);

-- create the IBMSNAP_TRAIL execution audit table
CREATE TABLE APPLY12.IBMSNAP_TRAIL (ENABLE SMALLINT NOT NULL,
DATA_SERVER CHAR(18) NOT NULL,BASE_OWNER CHAR (18) NOT NULL,BASE_TABLE
CHAR(18) NOT NULL,STMT_NUMBER SMALLINT NOT NULL,COPY_OWNER CHAR (18)
NOT NULL,COPY_TABLE CHAR(18) NOT NULL,LAST_DIFF_SEQUENCE CHAR(10) FOR
BIT DATA,RUNMODE CHAR(2) NOT NULL,STATUS SMALLINT NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,LASTRUN TIMESTAMP,FETCHED       INT NOT
NULL,FULLREFRESHED INT NOT NULL,INSERTED      INT NOT NULL,DELETED
INT NOT NULL,UPDATED       INT NOT NULL,REWORKED      INT NOT NULL,
INTERVAL_MINUTES INT NOT NULL,SQLSTATE CHAR(5),SQLCODE INTEGER,SQLERRP
CHAR(8),APPERRN CHAR(8),COPY_SERVER CHAR(18) NOT NULL,CNTL_SERVER CHAR(
18) NOT NULL,SQLERRM VARCHAR(70),APPERRM VARCHAR(760)) IN DPROPRTSPACE;

-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO APPLY12.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1  , 'DPROPRDB' , 'CENTRAL' , 'CUSTOMER_VIEWA' , 100 , 10 ,
'BRANCH12' , 'CUSTOMER_VIEW' , NULL , 'AO' , 0 , NULL,NULL, 12 ,'Y',
'Y','T', 4 ,'0101','TESTDB',NULL,'BRANCH_ID=12',NULL);

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 100 ,'A','CUSTOMER_ID','Y',
1 ,'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 100 ,'A','NAME','N', 2 ,
'NAME');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 100 ,'A','ADDRESS','N', 3 ,
'ADDRESS');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 100 ,'A','BRANCH_NUMBER','N',
4 ,'BRANCH_NUMBER');

-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO APPLY12.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1  , 'DPROPRDB' , 'CENTRAL' , 'CUSTOMER_VIEWC' , 101 , 10 ,
```

```
'BRANCH12' , 'CUSTOMER_VIEW' , NULL , 'AO' , 0 , NULL,NULL, 12 ,'Y',
'Y','T', 4 ,'0101','TESTDB',NULL,'BRANCH_ID=12',NULL);

-- edit template for an SQL-after statement to run at the COPY_SERVER
--* INSERT INTO APPLY12.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
--* BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
--* LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
--* INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
--* COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
--* VALUES (1  , 'TESTDB' , ' ' , ' ' , 200 , 10 , 'BRANCH12' ,
--* 'CUSTOMER_VIEW' , NULL , 'EI' , 0 , NULL,NULL,12,' ',' ',' ',0,
--* '0101','TESTDB',NULL,NULL,'<your SQL statement here>');
--*
-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 101 ,'A','CUSTOMER_ID','Y',
1 ,'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 101 ,'A','NAME','N', 2 ,
'NAME');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 101 ,'A','ADDRESS','N', 3 ,
'ADDRESS');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO APPLY12.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'TESTDB' , 'BRANCH12' , 'CUSTOMER_VIEW' , 101 ,'A','BRANCH_NUMBER','N',
4 ,'BRANCH_NUMBER');

-- the copy table does not yet exist
-- using REXX logic file SUBLOGIC.CMD

--* If you don't see: '-- now done interpreting...' then check your
--* REXX code
--*
-- in SUBLOGIC.CMD, about to create a copy table tablespace
CREATE TABLESPACE TS042308 MANAGED BY DATABASE
 USING (FILE 'C:\TEST\TS042308.F1' 500);

-- now done interpreting REXX logic file SUBLOGIC.CMD

-- create the copy table BRANCH12.CUSTOMER_VIEW
CREATE TABLE BRANCH12.CUSTOMER_VIEW(CUSTOMER_ID INTEGER NOT NULL,NAME
CHAR(18),ADDRESS  CHAR(20),BRANCH_NUMBER INTEGER,IBMSNAP_LOGMARKER
TIMESTAMP) IN TS042308;

-- create an index for the copy BRANCH12.CUSTOMER_VIEW
CREATE UNIQUE INDEX BRANCH12.CUSTOMER_VIEW ON BRANCH12.CUSTOMER_VIEW(
CUSTOMER_ID ASC);

--*
-- subscriber 'APPLY12' does not yet have a routing table defined
-- create the routing table, used to locate the CNTL_SERVER
CREATE TABLE APPLY12.IBMSNAP_ROUTING (CNTL_SERVER CHAR(18) NOT NULL,
```

```
TRY_CONNECT_TIME TIMESTAMP) IN DPROPRTSPACE;

CREATE UNIQUE INDEX APPLY12.XROUTING ON APPLY12.IBMSNAP_ROUTING (
CNTL_SERVER);

-- insert the CNTL_SERVER name into the routing table
INSERT INTO APPLY12.IBMSNAP_ROUTING (CNTL_SERVER) VALUES('TESTDB');

--* commit work at TESTDB
--*
COMMIT;

-- Satisfactory completion of SUB1 at 6:04pm
```

Now, it's time to add the SQL-after *delete* statement to the subscription. Before running the generated SQL, edit the SQL-after template in the file. Change this:

```
-- edit template for an SQL-after statement to run at the COPY_SERVER
--* INSERT INTO APPLY12.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
--* BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
--* LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
--* INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
--* COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
--* VALUES (1  , 'TESTDB' , ' ' , ' ' , 200 , 10 , 'BRANCH12' ,
--* 'CUSTOMER_VIEW' , NULL , 'EI' , 0 , NULL,NULL,12,' ',' ',' ',0,
--* '0101','TESTDB',NULL,NULL,'<your SQL statement here>');
--*
```

to this:

```
-- edit template for an SQL-after statement to run at the COPY_SERVER
INSERT INTO APPLY12.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1  , 'TESTDB' , ' ' , ' ' , 200 , 10 , 'BRANCH12' ,
'CUSTOMER_VIEW' , NULL , 'EI' , 0 , NULL,NULL,12,' ',' ',' ',0,
'0101','TESTDB',NULL,NULL,'DELETE FROM BRANCH12.CUSTOMER_VIEW C WHERE
C.CUSTOMER_ID IN (SELECT B.CUSTOMER_ID FROM BRANCH12.BRANCH B WHERE
B.BRANCH_NUMBER <> 12 OR B.IBMSNAP_OPERATION=''D''');
```

Once you've edited the SQL file to include the SQL-after delete statement, you can run the SQL file to define the subscription. Define similar subscriptions for ACCOUNT_VIEW and BALANCE_VIEW, each with an SQL-after delete statement to remove rows for customers who are no longer customers of branch 12. When all the subscriptions are defined, you can run an Apply process for subscriber APPLY12. This is a complex example, to be sure, but representative of techniques in use in advanced DPropR production environments. Realistically, an insurance company scenario would involve between two dozen and 50 (or more) copy tables, require more than a 2-way join to determine routing information in many instances, would involve replication to dozens or even hundreds of target databases, and so on.[1]

At this point in the scenario, assuming you've completed all the recommended steps, there should now be eight subscriptions for branch #12 (1 for the BRANCH table, 2 for the CUSTOMER table, 2 for the ACCOUNT table and 3 for the BALANCE table), though you logically only created *four* subscriptions. Functionally then, everything is in place. In practice, you may have a bit more work to do. It may be that your customers rarely change branches. If so, the pruning control table for the CENTRAL.BRANCH table will rarely receive updates. Subscriptions which only execute infrequently, or which rarely have change data to propagate, can cause CD and UOW table rows to not be pruned very quickly. For such infrequent subscriptions, further customization is necessary. (see "When pruning doesn't seem to be effective" on page 23) for additional information).

Outer join of two or more source tables with common primary key

Figure 28 illustrates one copy which is a materialized outer join (almost) of two source tables, each sharing the same primary key. This type of copy can be maintained by defining two subscriptions for the copy table. The copy table in this example must be created manually, as it contains more columns than either of the source tables T1 or T2. The user data columns must allow for null values, regardless of whether the source columns allowed null values, as an insert to either T1 or T2 will only result in a partial insert into T3. Furthermore, this table must be initialized manually, because, in this case, join views are *not* used.

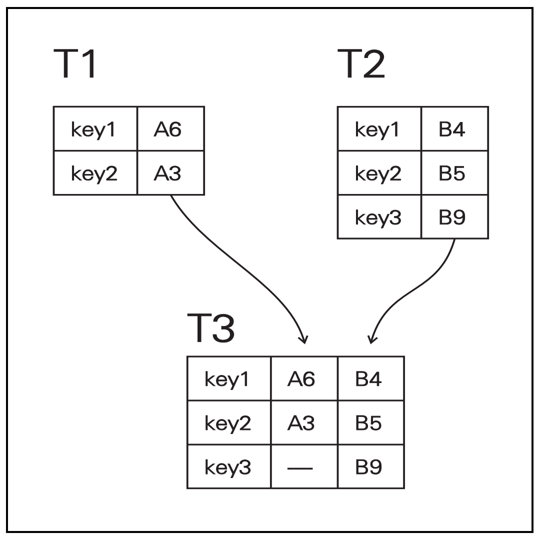

Figure 28 2-to-1 outer join copy.

[1] **Note:** If you have a copy of Capture/MVS prior to the August 25, 1995 refresh, when the named **CD_table** is actually a view definition, Capture/MVS will set CD_OLDEST_CHANGE to NULL, correctly indicating the gap condition, but Capture/MVS will not subsequently update the value of CD_OLDEST_CHANGE in the registration row when new change data is captured. You must reset CD_OLDEST_CHANGE to x'00000000000000000001' for your view registrations following a cold start of Capture/x. With the August 25 refresh, Capture/MVS maintains CD_OLDEST_CHANGE for view registrations.

The above scheme does not faithfully match true *outer join* semantics for deletes. For example, if a delete to T2 is followed by an insert into T2 with the same key value, the row in T3 will first be deleted and then reinserted but with just a value for column *B*, with column *A* left null, even in cases where there is a matching key in T1 and a known value for column *A*. This is because there is no join-view involved, merely two subscriptions independently updating the same copy table, using the same primary key.

Alternatively, you might consider taking advantage of the outer join capabilities of DB2 MVS V4. This would eliminate the need for manual initialization of the copy. If no deletes are processed against T1, the *inner* table in the previous example, you're in luck. If a row were deleted from T1, and a matching tow were already present in T3, then the delete would better be propagated as an *update*, setting column A to null. This would require additional registration and subscription complexity. Neither JOINREG nor the VREG function of the DPRTOOLS accept views defined using the new outer join syntax, either.

Target site UNIONS

Using DPRTOOLS in a data warehouse union scenario

note: this example has not been tested

Once again, we'll use table definitions from Figure 22 on page 60. In this scenario, we'll assume that we have two source databases, TESTDB and TESTDB2, and that information from the branch tables from both source databases will be consolidated into a single point in time copy in the DPROPRDB database.

Using the DPRTOOLS, the target table is going to be created in advance of the subscriptions, and two subset views created over the target table. Only when the copy and target views are in place, and the source tables registered, will the subscriptions be defined. The subscriptions will refer to the target view names, not the name of the copy table itself. The target views will restrict the Apply process to deleting only the appropriate rows when issuing a mass delete, which Apply does when beginning a full refresh of a point in time copy.

```
In database DPROPRDB, create the following:

CREATE TABLE WAREHOUS.BRANCH_UNION (CUSTOMER_ID IN NOT NULL,
BRANCH_NUMBER INT NOT NULL, IBMSNAP_LOGMARKER TIMESTAMP NOT
NULL)
```

```
CREATE VIEW TARGVIEW.BRANCH1 AS SELECT * FROM
WAREHOUS.BRANCH_UNION WHERE BRANCH_NUMBER=1

CREATE VIEW TARGVIEW.BRANCH2 AS SELECT * FROM
WAREHOUS.BRANCH_UNION WHERE BRANCH_NUMBER=2
```

If in your case, there is no column which distinguishes the source, such as BRANCH_NUMBER in the target views above, you can generate the necessary distinguishing column in your subscriptions and subsequently your copy one of two ways:

1. Create and register source views which contain a named column which is a different constant in each of the views. This is the cleanest approach, as once the views are created the rest of the details are taken care of using DPRTOOLS.
2. Add a computed column to each subscription containing a constant in the EXPRESSION field by adding an additional IBMSNAP_REF_COLS row for each subscription and the additional column in the copy.

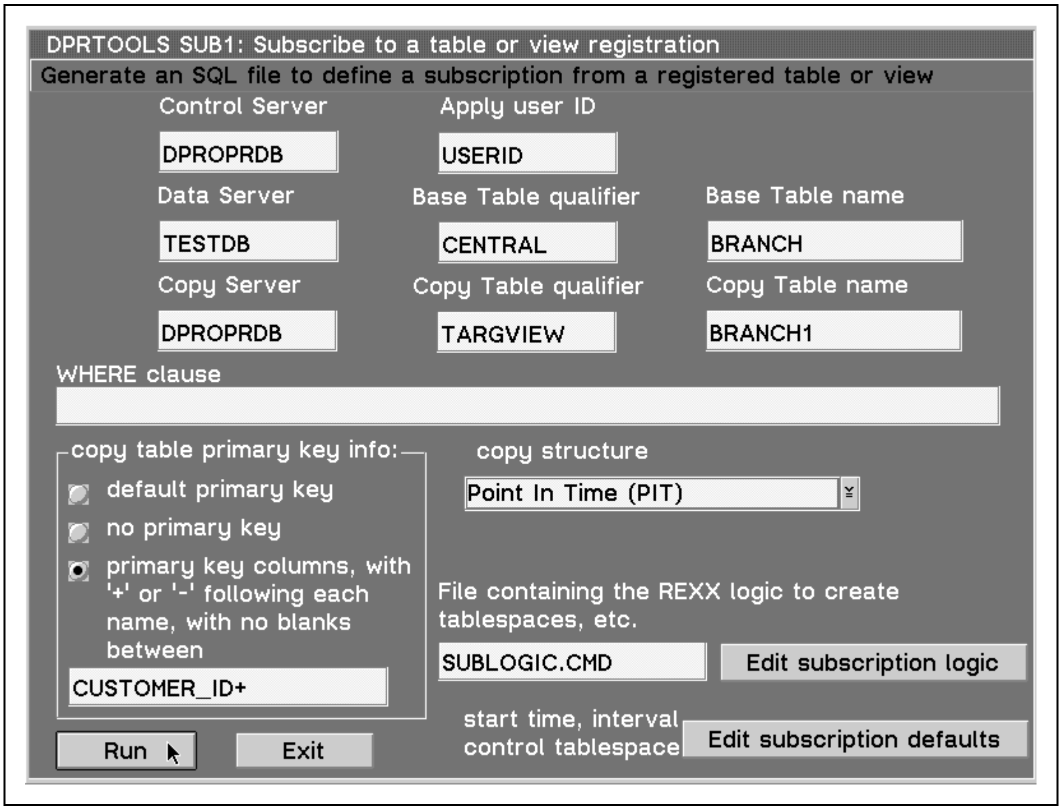

Figure 29 Defining half of the union; the subscription for data from branch #1.

In Figure 29, notice that the source database, the *data server* is TESTDB, the target database or *copy server* is DPROPRDB, and that the copy table is actually a view name, BRANCH1.

Here's the generated SQL file:

```
--* echo input: SUB1 DPROPRDB USERID TESTDB CENTRAL BRANCH DPROPRDB
--* TARGVIEW BRANCH1 CUSTOMER_ID+ PIT SUBLOGIC.CMD
--*
-- SUB1 test version 04/05/96
-- using REXX defaults file SUB1DFLT.CMD

--* If you don't see: '-- now done interpreting...' then check your
--* REXX code
--*
-- now done interpreting REXX logic file SUB1DFLT.CMD

--* connect to the data_server
--*
CONNECT TO TESTDB;

-- current USERID=USERID   DATA_SERVER alias=TESTDB   16 Apr 1996 7:58pm
-- using SUBLOGIC.CMD as the  REXX logic filename
```

```
--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000003(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('DPROPRDB','TARGVIEW',
'BRANCH1','DPROPRDB',4);

-- commit work at server TESTDB
COMMIT;
-- switch context to server
CONNECT TO DPROPRDB;
-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1 ,'TESTDB','CENTRAL','BRANCH',100,10,'TARGVIEW','BRANCH1',
NULL,'AO',0,NULL,NULL,12,'Y','Y','T',4,'0101','DPROPRDB',NULL,NULL,
NULL);

-- edit template for an SQL-after statement to run at the COPY_SERVER
--* INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
--* BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
--* LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
--* INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
--* COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
--* VALUES (1 ,'DPROPRDB',' ',' ',200,10,'TARGVIEW','BRANCH1',NULL,
--* 'EI',0,NULL,NULL,12,' ',' ',' ',0,'0101','DPROPRDB',NULL,NULL,
--* '<your SQL statement here>');
--*
-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','TARGVIEW','BRANCH1',100,'A','CUSTOMER_ID','Y',1,
'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','TARGVIEW','BRANCH1',100,'A','BRANCH_NUMBER','N',2,
'BRANCH_NUMBER');

--* commit work at DPROPRDB
--*
COMMIT;

-- Satisfactory completion of SUB1 at 7:58pm
```

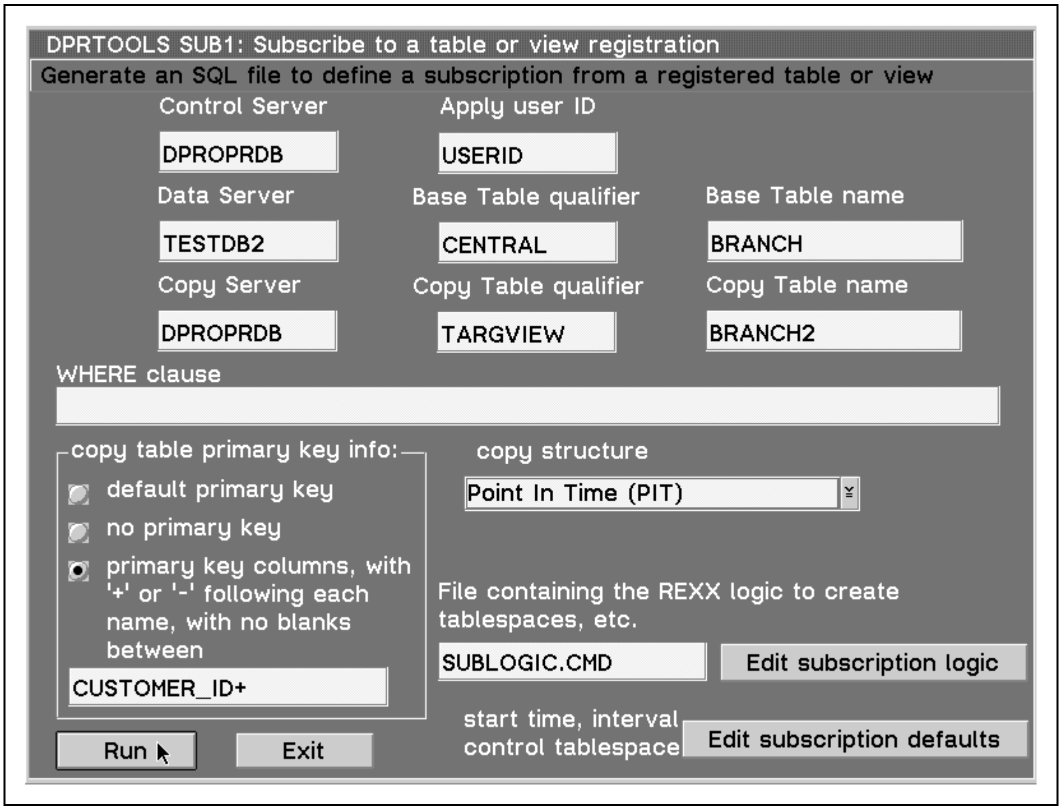

Figure 30 Defining half of the union; the subscription for data from branch #2.

In Figure 30, the *data server* is changed to TESTDB2, and the *copy table* is changed to the second target view name, BRANCH2.

```
--* echo input: SUB1 DPROPRDB USERID TESTDB2 CENTRAL BRANCH DPROPRDB
--* TARGVIEW BRANCH2 CUSTOMER_ID+ PIT SUBLOGIC.CMD
--*
-- SUB1 test version 04/05/96
-- using REXX defaults file SUB1DFLT.CMD

--* If you don't see: '-- now done interpreting...' then check your
--* REXX code
--*
-- now done interpreting REXX logic file SUB1DFLT.CMD

--* connect to the data_server
--*
CONNECT TO TESTDB2;

-- current USERID=USERID   DATA_SERVER alias=TESTDB2  16 Apr 1996 7:58pm
-- using SUBLOGIC.CMD as the  REXX logic filename
--* record the subscription in the pruning control table  at the data
--* server
INSERT INTO CENTRAL.PC0000000000000003(COPY_SERVER,COPY_OWNER,
COPY_TABLE,CNTL_SERVER,COPY_STRUCTURE)VALUES('DPROPRDB','TARGVIEW',
```

```
'BRANCH2','DPROPRDB',4);

-- commit work at server TESTDB2
COMMIT;
-- switch context to server
CONNECT TO DPROPRDB;
-- create a new subscription row in IBMSNAP_REF_CNTL
INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
VALUES (1 ,'TESTDB2','CENTRAL','BRANCH',100,10,'TARGVIEW','BRANCH2',
NULL,'AO',0,NULL,NULL,12,'Y','Y','T',4,'0101','DPROPRDB',NULL,NULL,
NULL);

-- edit template for an SQL-after statement to run at the COPY_SERVER
--* INSERT INTO USERID.IBMSNAP_REF_CNTL(ENABLE,DATA_SERVER,BASE_OWNER,
--* BASE_TABLE,STMT_NUMBER,PRIORITY,COPY_OWNER,COPY_TABLE,
--* LAST_DIFF_SEQUENCE,RUNMODE,STATUS,IBMSNAP_LOGMARKER,LASTRUN,
--* INTERVAL_MINUTES,COPY_CONDENSED,COPY_COMPLETE,COPY_CONSISTENT,
--* COPY_STRUCTURE,ARCH_LEVEL,COPY_SERVER,EXPIRES,PREDICATES,SQL_STMT)
--* VALUES (1 ,'DPROPRDB',' ',' ',200,10,'TARGVIEW','BRANCH2',NULL,
--* 'EI',0,NULL,NULL,12,' ',' ',' ',0,'0101','DPROPRDB',NULL,NULL,
--* '<your SQL statement here>');
--*
-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','TARGVIEW','BRANCH2',100,'A','CUSTOMER_ID','Y',1,
'CUSTOMER_ID');

-- create a new row in IBMSNAP_REF_COLS
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','TARGVIEW','BRANCH2',100,'A','BRANCH_NUMBER','N',2,
'BRANCH_NUMBER');

--* commit work at DPROPRDB
--*
COMMIT;

-- Satisfactory completion of SUB1 at 7:59pm
```

3-tier consolidation scenario

What should you do when you want to have a 3-tier consolidation scenario, where the data is generated at tier #3, is gathered into regional unions at tier #2, and finally brought together in a global union at tier #1? In this case, you can consider copying from tier #3 into point in time copies at tier #2, registering these copies as *user tables* at tier #2, capturing each change again at tier #2, then finally subscribing from what appear to be user tables at tier #2 for the final copying up to tier #1.

By recapturing the updates at tier #2, the original tier #3 sequence numbers are replaced with tier #2 sequence numbers, thereby closing the data loss exposure, as explained in the note above. Some of this setup must be customized, including:

1. The point in time copies at tier #2 must be created with the change data capture attribute. Edit the create table statements to include DATA CAPTURE CHANGES.

2. The point in time copies at tier #2 must be registered as user tables, not point in time copies. The table registration feature in DPRTOOLS will do this automatically, omitting the IBMSNAP_LOGMARKER column from the tier #2 registrations.

Defining views over CCD tables

November 14, 1995 Note: this technique is for experts only, and creates a registration which may require additional migration effort in the future.

While it is possible to register a view of a CCD table, with IBMSNAP_CD_CNTL.BASE_STRUCTURE=3, this registration is not a usable source for an Apply process. A DPropR Apply process will always issue a LOCK TABLE statement against a CCD table, whether the CCD table is a source or a target, in order to properly serialize between the Apply process which is writing to the CCD table and any Apply processes which may read from the CCD table. This is a problem for view registrations, as SQL does not support references to views in a LOCK TABLE statement.

So, for views over complete CCD tables, the trick is to register the CCD view not as a CCD table, but something else. Actually, a *couple* of somethings. Though the source view contains the IBMSNAP_OPERATION, IBMSNAP_INTENTSEQ, IBMSNAP_COMMITSEQ and IBMSNAP_LOGMARKER columns matching the CCD structure, you'll take the name of the CCD table to set the values of both BASE_TABLE and CD_TABLE. Additionally, you'll set CD_CONSISTENT to 'C', for *convergent consistency*, BASE_STRUCTURE=1, as if the view was a *user table*, BASE_COMPLETE='Y', CD_COMPLETE='N', and so on. Though there is just one view, the Apply process will be fooled into believing it is a base table some of the time and a CD table some of the time.

The Apply process will not issue the LOCK TABLE statement when it believes the source is a user table, or when it believes the source is a convergent consistent change data table.

Work-around for the before-image column predicate restriction

Define a view for the user table which includes the name of the before image column in the change data table, with a NULL used as the column value in the subselect defining the view. As "NULL" cannot be used directly in the view definition, you must first define a dummy table with a single column and one NULL row, joining with this table whenever a null is required.

Register your view as a base table. Complete the view registration by referencing the change data and other physical tables named in the registration of the physical user table. With the column now "included" in the base table (assuming subscriptions refer to the source through the view registration), you may refer to the before image column in your SQL predicates for subscriptions.

To PUSH or to PULL

There are four logical locations from the perspective of the apply program:

- LOCAL SERVER where the routing table is stored and the default server for Apply/x
- CNTL SERVER where to store the REF_CNTL, REF_COLS and _TRAIL tables
- DATA SERVER where the source table is; the source database
- COPY SERVER where you want the copy to be; the target database

Normally, the apply program will run on the same platform (operating system image) that is also running the DBMS that is both the LOCAL SERVER and COPY SERVER. This is not an architectural restriction, however.

The Apply process can be run remotely from the source, remotely from the target, or remote from *both* the source and target, allowing for *pull*, *push* and *indirect* configurations.

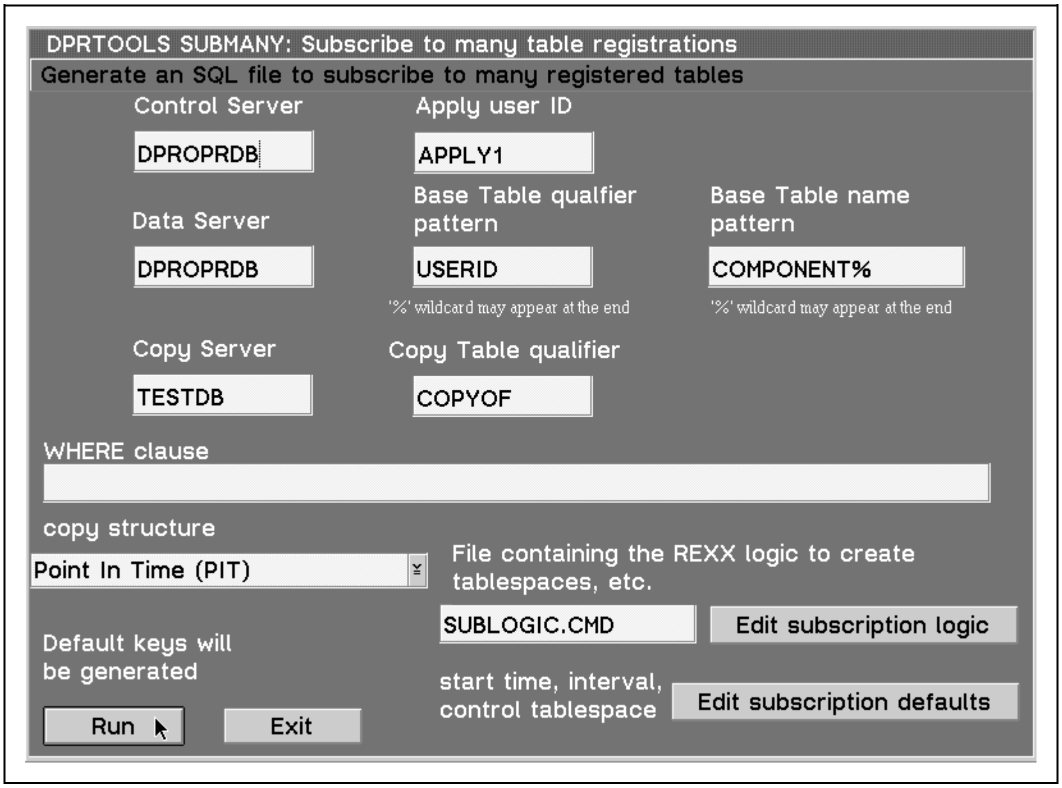

Figure 31 Setting up an efficient *push* configuration, assuming that the Apply process is run local to the DPROPRDB database.

A pull configuration (see Figure 32) is typically between 10 and 30 times more efficient than a push configuration, due to the block fetch optimization associated with remote fetch processing.

If the update rate is very low, a push configuration can be desirable (see Figure 31). If you want to reduce delay of propagation, you'll need to define your subscriptions with low INTERVAL_MINUTES values. This will increase overhead because the subscription control tables will be accessed more frequently. You can shift this burden away from a highly loaded target server, and keep this "polling for work" activity off the network, by running the Apply process (and keeping the subscription control tables) local to the source.

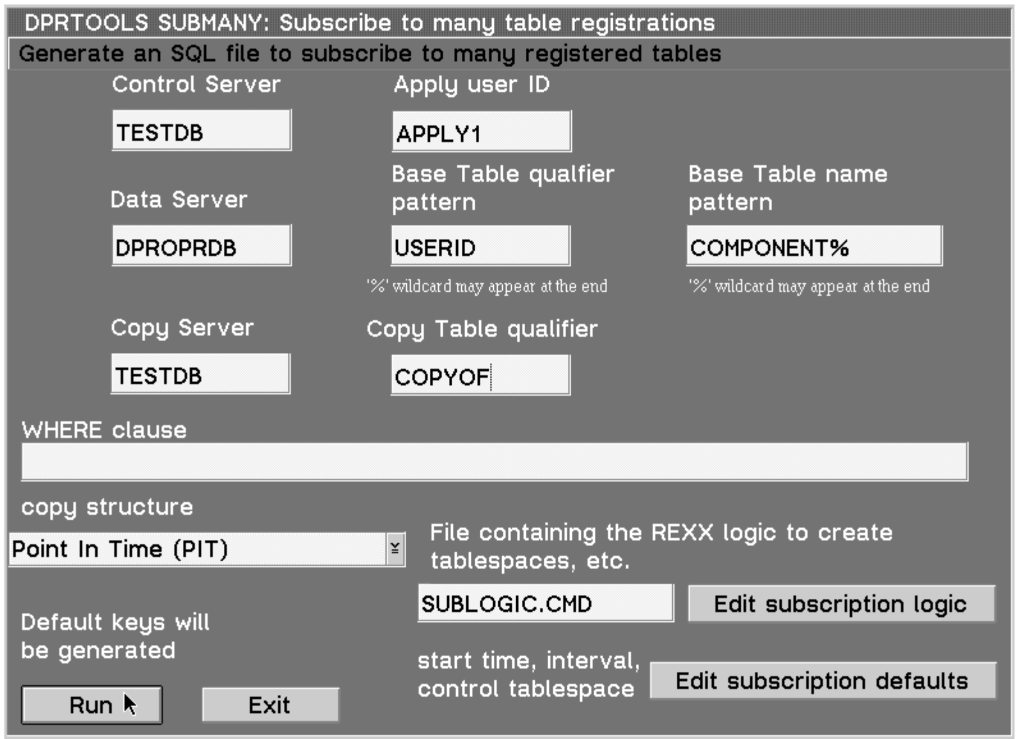

Figure 32 Setting up an efficient *pull* configuration, assuming that the Apply process is run local to the TESTDB database.

Two-way propagation

Note: there is at yet no known successful use of this procedure.

It is most likely that two-way propagation is not acceptable for your application, for reasons which you may not yet be aware of. Please read the InfoDB reprint "Update Replication: What Every Designer Should Know", listed in the references section, before proceeding.

Because two-way propagation, in this case specifically *asynchronous replication*, is such a tricky area, the administration will not allow you to set up circular copy definitions of any type. To safely set up the definitions yourself, you will want to consider the problem of *update collision* -- when two or more replicas update the same row essentially simultaneously. The easiest way to address the collision problem is to avoid it, by ensuring that one or both of the following conditions always holds:

- updating transactions at each site, though they may read the entire table, only ever update mutually exclusive fragments

- transactions may update any part of the table, but each site is limited to making updates during exclusive updating time intervals, such as eight hour shifts in New York and Tokyo.

Let's use a New York/Tokyo scenario to show one of the many ways to approach a replication configuration. Assume that DB2, Capture/MVS, and Apply/MVS are installed at both locations. Create a new table in New York with the IBMSNAP_LOGMARKER column required for *point in time* tables. With this technique, no other additional columns are needed. Define subset views for your applications if you wish to hide the IBMSNAP_LOGMARKER column from your applications; it must not be hidden from Apply/MVS. Register this table for copying and change capture. Create a similar table in Tokyo and also register this table for change capture also. As a safety measure, disable the *fullrefreshenable* control flag for each replica in the IBMSNAP_CD_CNTL tables at each site.

There are two ways for you to initially synchronize the two tables and all methods are "tricky". First, let's discuss the subscription definitions and the ongoing propagation activity. Later, we'll go back and discuss how the two subscriptions must be started.

For each of your Apply/MVS programs, choose a unique user ID that will not be used by any other updating SQL application on either system. For this example, we'll use NYUSER and TOKUSER for the subscriber IDs used in New York and Tokyo, respectively. These user IDs will be used to distinguish propagated-in updates from ordinary local updates. This will be important in eliminating the possibility of circular propagations.

Set up one IBMSNAP_REF_CNTL table with a subscription definition to copy data from New York to Tokyo. For the subscription to maintain the copy in New York, include predicates (one or two, depending on whether you use one **¬=** or two **<** and **>** operators) to exclude updates made by

IBMSNAP_AUTHID = 'NYUSER', which will only be associated with updates made originally to the New York database and recently propagated to Tokyo. Set up a corresponding subscription to maintain the copy in Tokyo; exclude updates where IBMSNAP_AUTHID = 'TOKUSER'. You'll notice that the IBMSNAP_AUTHID column comes from the ASN.IBMSNAP_UOW table. Apply/MVS will reference this table in a join with the CD table, when propagating *changes* to a copy. This has two implications:

> 1) If you use CCD tables in this scenario, the CCD tables must include the IBMSNAP_AUTHID column -- refer to the "Auditing" section on page 86.
>
> 2) You must either ensure that Apply/MVS does not perform a full refresh to initialize one of the *peer* point in time copies, or you must only add the exclusion predicates mentioned above *after* Apply/MVS has completed the full refresh. Rely on the iteration control (ENABLE = -1) to isolate the initial full refresh, if necessary. The predicates require this special consideration as the ASN.IBMSNAP_UOW table reference is not generated by Apply/MVS during a full refresh cycle, only a differential refresh cycle and only when sourcing from a CD table. When the source is not a CD/UOW join then, the reference to IBMSNAP_AUTHID will be in error, unless you satisfy implication (1) above.

Now that you understand that you'll be using predicates that refer to a column in the ASN.IBMSNAP_UOW table, and what that means, we'll discuss your options for initially synchronizing the two tables in New York and Tokyo.

CASE1: If you use a load utility to load one copy from the other, or to load *both* tables, you'll need to use the technique described in the "Automating the large copy load" section on page 34 to ensure that neither Apply/MVS program performs an initial full refresh of a copy.

CASE2: If you want to let Apply/MVS in Tokyo, say, initialize the copy there from the corresponding table in New York, you will have to use the large copy load technique to disable any full refresh of the copy in New York, and define the exclusion predicate(s) in the subscription for the copy in Tokyo only after that copy is initialized with the data from New York. You can use a -1 in the IBMSNAP_REF_CNTL.ENABLE column to allow the subscription to run just one time to perform the initial full refresh, then add the necessary exclusion predicate(s), then set IBMSNAP_REF_CNTL.ENABLE to +1 to let it run freely.

Finally, begin using the replicas, always complying with one of the two update restrictions above. Once the setup is running, be sure not to use a load (or recover) utility again as any loaded rows will bypass the change capture mechanism.

Note: each of the replicas must have the physical structure of a point in time copy, have the COPY_STRUCTURE value =4 for *point in time* table in the respective refresh control table, but have a BASE_STRUCTURE value =1 for *user table* in each respective change data control table.

Work-around for copy-owner naming restriction

Currently, the DPropR/2 administration component will use the subscriber's user ID as the *owner* qualifier when creating a new copy table. There are a number of bypasses available to you, should you need to vary the table CREATOR qualifier for your copy tables.

Keep in mind that each copy must be uniquely identified in order for the pruning algorithm employed by Capture/MVS to work correctly. If two copies have the same name, their integrity will be compromised, as some of the changes that should be applied to these copies will be pruned prematurely. Whatever bypass method you choose, you must take responsibility for maintaining the uniqueness of the copy table names. In simple configurations, this should not be an issue. For configurations with many copy servers, it would certainly be an issue.

You may want to change the unique indices on the pruning control tables to include the COPY_SERVER column, if that will help you achieve uniqueness of copy names in the pruning control tables (the DPRTOOLS include the COPY_SERVER column when creating a pruning control table index).

USE VIEWS. This is a general and safe approach. Set up your subscriptions using DPropR/2, but set the "activate" button in the frequency page to "no". Then, issue "CREATE TABLE LIKE.." statements to create your own set of tables with the same structure as the tables created by DPropR/2 but having the owner qualifiers you prefer. Also create an index on the primary key, just as DPropR/2 would do. DROP the copy table created by DPropR/2, then create a view with the name of the just-dropped copy table which refers to the table with the owner qualifier you prefer. The Apply/MVS program will not care that it is updating through a view, and it will happily be able to manage the copies according to the view names. You could batch up all the CREATE TABLE, CREATE INDEX, CREATE VIEW statements and run them using SPUFI or other SQL tool. Finally, activate all the subscriptions. You can do this through the notebook interface or by setting the values of IBMSNAP_REF_CNTL.ENABLE to +1.

You might prefer dropping the copies DPropR/2 created as the first step, if there is insufficient storage to have the two sets of tables existing simultaneously.

EDIT THE IBMSNAP_REF_CNTL, IBMSNAP_REF_COLS TABLEs. If you can enforce uniqueness of copy table names yourself, you could:

1) Define your subscriptions and keep them deactivated, as above.
2) Create your own copy tables and indexes, as above. Be sure the table structure matches the table created by DPropR/2.
3) Drop the copy tables created by DPropR/2.
4) Instead of defining views, replace the COPY_OWNER values in the appropriate IBMSNAP_REF_CNTL table with the owner qualifier of the tables you've just created.
5) Also replace the COPY_OWNER values in the appropriate IBMSNAP_REF_COLS table with the owner qualifier of the tables you've just created.
6) If you'll be using the same copy table names on two or more copy servers, then you'll want to redefine the unique index on the pruning control table at the data server to include the COPY_SERVER column. That way, each subscription will retain a unique row in the pruning control table and the progress of the individual subscriptions will still be tracked correctly.
7) Activate your subscriptions.

USE THE DPRTOOLS

You'll need to redefine the unique indexes on any existing pruning control tables to include the COPY_SERVER column. New pruning control table indices created by the DPRTOOLS will include the COPY_SERVER column. Once the indices are set, you can create subscriptions using any copy table qualifier you wish.

Condensing utility

Complete histories, by definition, will consume more and more disk storage over time. You might wish to keep only six month's worth of historical data. Assuming you have a COMPLETE=Y, CONDENSED=N consistent change data (CCD) table defined, you may write your own SQL to carefully condense all rows older than six months. This is not the same as *deleting* all rows older than six months. To satisfy the COMPLETE=Y attribute condition, you will need to always leave the most current row for each key (even rows representing *deletes*) in the CCD table.

Scheduling of all refreshes from your MVS dependency management system

Because of application semantics such as *end of day processing*, it is often important to make copy refresh activity dependent upon the completion status of batch update applications. To do this, either code additional SQL into your existing applications, or write a small new application to update the ENABLE field of the particular IBMSNAP_REF_CNTL table(s). This field can be used as an on/off switch, or set to a negative number for |-n| fixed number of refreshes. This way, your batch job dependency scheduler can hold subscription refresh activity until all dependencies are satisfied.

Auditing

Three of the IBMSNAP_UOW columns, IBMSNAP_AUTHTKN, IBMSNAP_AUTHID and IBMSNAP_LOGMARKER provide the DB2 correlation ID, primary SQL ID and approximate commit timestamp, respectively, associated with the original updating transaction.

You will not find these columns available for selection in the subscription notebook. If your are subscribing indirectly to the original source table, through a CCD staging table which is defined as a *base table*, you may enter these column names via the "create column" feature of the columns page of the subscription notebook. Otherwise, you must define the auditing subscription manually (see page ?).

Along with these transaction-related columns, you will of course include the columns from the *user table* being audited. Anticipating an auditing requirement, you will want to consider registering both the *before* and *after* change image of every column, and certainly you must register the user table enabling both refresh and *update* copying, to be sure that the change data capture mechanism is put in place.

Additionally, you will want to be aware of some special handling that occurs for local CCD table subscriptions. The first such subscription is assumed to be a staging table for other subscriptions and not an auditing table. This first CCD table is placed first in the apply program search order, in front of the CD table in the choice of alternatives to accessing the *base table*, in this case specifically, a *user table*. If you intend to have your apply program append the auditing CCD table only infrequently, you may want to remove its association as a preferred alternative source to the user table. Why? Because other subscriptions will be regulated by the advancing of the CCD table. If the CCD table is updated only every six hours, say, and is named in the CCD_OWNER, CCD_TABLE columns of the IBMSNAP_CD_CNTL table row for the user table being audited, than other subscriptions to the user table will be subject to six hour currency lag, regardless of their individual INTERVAL_MINUTES, or *frequency*, values. So, you may want to set CCD_OWNER and CCD_NAME to *null* to take the CCD table out of the search order. If the audit-CCD subscription is not the first local CCD subscription for this user table, then it will not be associated as an alternative source. In that case, you may ignore the discussion in this paragraph.

To be sure your CCD table will have the attributes you desire, have a look at Figure 2 on page 5 to see if the defaults assumed by DPropR/2 will be correct, and if not, create your subscription with the *activate* button on the frequency page of the subscription notebook set off (IBMSNAP_REF_CNTL.ENABLE = 0). Then, manipulate the IBMSNAP_REF_CNTL table values for CCD_COMPLETE as appropriate. Then, set the ENABLE field to 1, and your CCD table will be maintained as you wish.

Work-around for the 17-byte-unique column name restriction

It is not possible to capture both *before* and *after* images of columns if the first 17 bytes of the source table columns are not unique within the table. This is because of the truncation that occurs when prefixing a column name to indicate that it is a *before image* column.

To work-around the restriction, first unload the existing table. Recreate the table under a different name and with 17-byte unique column names, then load the data back into the new table. Define a view with the name of the old table and old table column names on the new table, then rebind the applications. Register the new table and freely specify *before* and *after* image column names during registration.

If you like, also register views of the new table and its change data table, for subscriptions which need not include any *before image* columns and would better be defined using the original column names.

Detecting DB2 MVS Utility Operations

To determine if a LOAD or RECOVER utility has been run which may result in the invalidation of your differentially maintained copies, issue the following query against the DB2 catalog and DpropR control information:

```
SELECT BASE_OWNER, BASE_TABLE, ICTYPE, TIMESTAMP
FROM SYSIBM.SYSCOPY S, SYSIBM.SYSTABLES T, ASN.IBMSNAP_CD_CNTL C
WHERE BASE_OWNER = T.CREATOR AND BASE_TABLE = T.NAME
AND T.DBNAME = S.DBNAME AND T.TSNAME = S.TSNAME
AND ICTYPE IN ('P','R','S','Y','Z','T')
AND T.TYPE = 'T' AND DATACAPTURE = 'Y'
ORDER BY TIMESTAMP DESC;
```

Querying recent Apply activity

This query from Thierry Jouseau of Apicil/Arcil, who uses it to check up on his subscriptions:

```
SELECT ENABLE, DATA_SERVER, BASE_OWNER, BASE_TABLE,
STMT_NUMBER, COPY_OWNER, COPY_TABLE, LAST_DIFF_SEQUENCE,
RUNMODE, STATUS, IBMSNAP_LOGMARKER, LASTRUN, FETCHED,
FULLREFRESHED, INSERTED, DELETED, UPDATED, REWORKED,
INTERVAL_MINUTES, SQLSTATE, SQLCODE, SQLERRP, APPERRN,
COPY_SERVER, CNTL_SERVER, SQLERRM, APPERRM
FROM SM.IBMSNAP_TRAIL
WHERE BASE_TABLE LIKE '&TABLE'
AND DATE(LASTRUN) >= '&JOUR'
AND TIME(LASTRUN) > '16.00'
AND FETCHED > 0
ORDER BY LASTRUN DESC
```

When potential DB2 registrars, subscribers not listed

In some cases, user IDs known to the source operating platform will not appear under the object listings for *(DPropR) Registrars to be authorized* or *(DPropR) Subscribers to be authorized*. Only user IDs which appear in the DB2 catalog are available to be listed. If a particular user ID does own an object or is a *grantee*, it will be available to be listed, otherwise it will not. You have three possible actions available to you:

- create a table with the user ID as the OWNER
- grant a specific privilege to the user ID
- manually insert a row into ASN.IBMSNAP_AUTH at the data_server, indicating the level of DPropR authority for this user ID.

  Note: this technique may lead to trouble if this user ID does not otherwise have sufficient authority or privileges to successfully perform the DPropR/2 actions available to a *registrar* or *subscriber*, as applicable.

Application Backout

For IMS/TM and CICS customers, the value of IBMSNAP_AUTHTKN will provide useful information for tying a specific transaction program with DB2 updates. Say, for example, that you want to back out the effects of transaction program P1 executions from 10:00am until 12:00pm. Assuming that you are capturing both before and after images for each update, it is possible to *compensate* for the effects of transaction by reversing the update, that is, assigning the *before* values back into each row updated by the transaction during the period.

First, search for the executions of P1 in the values of ASN.IBMSNAP_UOW.IBMSNAP_AUTHTKN for transactions committed during the 10:00..12:00 time interval, using date/time predicates referencing ASN.IBMSNAP_UOW.LOGMARKER . Join these values with each of the CD tables on matching IBMSNAP_UOWID values. For each row in the answer set of the resulting joins, generate a compensating SQL statement, re-inserting deleted rows, deleting inserted rows, and updating updated rows with the prior, before-image, values.

Determining change capture program lag

Issue the following query to approximate the lag of the change capture program:

```
SELECT
HOUR(CURRENT TIMESTAMP - MAX(IBMSNAP_LOGMARKER)) AS LAG_HOURS,
MINUTE(CURRENT TIMESTAMP - MAX(IBMSNAP_LOGMARKER)) AS LAG_MINUTES
FROM ASN.IBMSNAP_UOW
```

The result of the query will over-estimate the present lag of the change capture program.

Automating the stopping of Apply/2

As you may have discovered, there is no **STOP** command for Apply/2. You may desire such a command if you have need of automated scripts for stopping and starting your OS/2 software at particular times.

You may write your own program to gracefully stop Apply/2. Your program would need to issue an SQL update statement to set the value of IBMSNAP_REF_CNTL.ENABLE to 2, in the global control record, where COPY_TABLE is blank. You may then execute this small SQL application from your control script to stop Apply/2.

Determining copy currency

Issue one of the following queries:

> SELECT MAX(IBMSNAP_LOGMARKER) FROM copy; *this timestamp is from the SQL clock at the data_server*
>
> SELECT LASTRUN FROM IBMSNAP_REF_CNTL WHERE COPY_OWNER = subscriber-id AND COPY_TABLE = copy-tablename; *this timestamp is from the SQL clock at the control server*

You can use the REPORTS reporting facility of DPRTOOLS to list the status of all subscriptions from a specific data server.

Regulating Full-refresh activity

If you would like to ensure that your data server will not honor requests for initializing or reinitializing of new copies, which may result in unusually large answer sets and high network load, you may temporarily disable all such activity by issuing the following SQL statement:

```
UPDATE ASN.IBMSNAP_CD_CNTL
SET FULLREFRESHENABLE  = 0;
```

You may disable full refresh for specific base tables only, by supplying the appropriate predicate(s). You may later restore normal operation by issuing the following SQL statement:

```
UPDATE ASN.IBMSNAP_CD_CNTL
SET FULLREFRESHENABLE  = 1;
```

Generating Anonymous Test Data

You may wish to supply your DB2/2 development environments with representative DB2 test data, but for ethical or legal reasons you are restricted from distributing confidential DB2 data. You have a number of options available, among them:

1. set up views to substitute, say, an individual's given name with the name of a tree, such as *poplar*, *birch*, *walnut*, and so on. You could even retain gender distinction by using appropriately mapping male and female names to softwoods and hardwoods, for example. (see page ?). The subscriptions to supply the test data would refer only to the registered substitution-view sources.

2. A more secure scheme would be to define a DB2 EDITPROC on either a change data or consistent change data table in DB2. Within the EDITPROC, you would include a function to manipulate sensitive data columns using some randomizing scheme. The CD or CCD table would receive only randomized, but otherwise "realistic" test data.

   The best scheme might be to define a DB2 CONDENSED=Y, COMPLETE=N CCD subscription, registering this table for copying as DB2 base table. Define the DB2 EDITPROC for this copy only. That way, the balance of your copies would continue to receive genuine, albeit sensitive data. DB2/2 subscriptions of this DB2 CCD base table then, would never include data directly from the user table containing the genuine, sensitive data. Define the copies in DB2/2 to also be COMPLETE=N, CONDENSED=Y CCD tables. If necessary, you could define views on these tables to filter out the IBMSNAP_ overhead columns. The copies would start out empty but grow over time, as the original DB2 data is updated. Subscriptions would supply these test tables with a steady flow of representative and changing test data.

In either scheme above, only de-sensitized data flows over the network.

Automated pruning of IBMSNAP_TRAIL

Each subscriber owns one or more IBMSNAP_TRAIL tables, depending on the control server option chosen, which can grow large enough to become a problem. Some customers write small SQL applications to delete old rows from these tables. You can provide your Apply/x with definitions to delete old rows *automatically*.

Add such a definition by first composing your delete statement for the audit trail table with the appropriate age criteria, using the time and date functions of SQL (DATE, MONTH, etc.) and current time information in the CURRENT TIMESTAMP special register, as appropriate. Take this delete statement and insert it into a new row in the IBMSNAP_REF_CNTL table in the SQL_STMT column. Also in this new row, refer to the particular audit trail table in the COPY_SERVER, COPY_OWNER and COPY_TABLE columns, and specify the "EI" RUNMODE, the ENABLE as 1, the ARCH_LEVEL as ’0101’, the

INTERVAL_MINUTES to indicate how often to execute your delete statement, and all other NOT NULL columns with defaults of either blank or numeric zero, as appropriate.

Migrating from test to production

This section will introduce you to three options, depending on which tools you prefer:

1. If you're using DPropR/2 to define your registrations and subscriptions and plan ahead, you can trace your DataHub/2 sessions, in order to save the SQL statements that DPropR/2 generated and ran on your test system. Registrations and subscriptions, after all, are just series of SQL statements. Then, using ASNBATCH to post-process the traced SQL, you can re-run the original statements against the production database, after possibly editing the file.

2. However you managed to define your registrations and subscriptions, using DPropR/2, the DPRTOOLS or by typing SQL directly, you can use the PROMOTER and PROMOTES functions of the DPRTOOLS to reverse-engineer your registrations and subscriptions, respectively. These tools will produce files with the DDL to create CD and copy tables, etc., and the inserts into the appropriate DPropR tables. You then have the opportunity to edit these SQL files before running them, just as when using ASNBATCH with DPropR/2.

3. If you're using the DPRTOOLS to setup your test environment, you can simply save the SQL files produced when defining your registrations and subscriptions. Once saved, you can then easily edit and rerun against another database, just as with the techniques mentioned above.

Managing magnetic and optical storage for the same DB2 table

The following design has not been thoroughly evaluated but is being discussed within the context of a medical records application. The scheme for managing optical storage and magnetic storage within the scope of a single table comes out of discussions during a customer briefing on 10/31/94. One of the customer's technical staff suggested using DB2 partitioning to achieve a single table image of current data on magnetic disk and non-current data on optical disk. This managed storage scheme would be functionally transparent to applications, with queries against current data hopefully falling into the same performance class as their queries today and with queries referencing non-current data falling into a lower performance class. The DB2 partition and tablespace size limits were said not to be restrictive for this application.

The customer maintains patient records in historical format, with the most current patient history records available online, stored on magnetic disk. Records which are not so current are today archived and unavailable for query. The customer would prefer a transparent type of "archiving", whereby the data is still available for query, albeit at the lower performance of lower cost optical disks.

The proposed DPropR solution relies on DB2 table partitioning, just as the customer suggested. First, we'll assume that each application table is in turn unloaded and recreated with a different name and with five additional columns. After reloading the data into the new tables, views are defined with the same name and structure of the previous tables and the applications rebound, thus hiding the new columns from the applications. Note that with this scheme, the application code is not changed, just rebound to use the subset views.

Four of the five new application table columns are control columns expected in every DPropR CCD table. The customer does not need these columns, but Apply/MVS insists on these columns being in the target table. The fifth new column is an integer column, "partition_key", which will serve as the partitioning key. This column is defined as NOT NULL WITH DEFAULT, with the convention being that all current rows are stored in partition 0, which is backed by magnetic disk storage. As the applications only insert through subset views which do not contain this column, all current data is automatically placed in partition 0.

The semantics of the DELETE statement are changed for this table, so that deletion of a row in partition 0 results in an asynchronous migration of that row from partition 0 to another partition in the same table; a partition backed by optical disk storage. The timing of such migrations could be planned for off-peak periods, hiding the momentary loss of data from a table resulting from the asynchronous row migrations from one partition to the next.

To accomplish the migration, change capture is enabled for the application table (which now has the structure of a DPropR history table, or consistent change data -- CCD table) and a subscription is defined with the source and target being the very same table. By manipulating the Apply/MVS control information, Apply/MVS is led to believe that the source is a "user table" and that the target is a non-complete non-condensed CCD table, so that it is only ever appended, using SQL INSERTs.

Additionally, all insert and update operations must be filtered out in the subscription predicate so that only deleted rows are picked up by Apply/MVS and inserted into the target -- the very same table. In other words, the subscription filters out change data related to insert or update operations against the table.

To accomplish the migration from partition 0, the partition_key column can be defined in the subscription definition as a calculation, such as using the YEAR function against:

a) the timestamp of the original delete operation,

b) the YEAR of the CURRENT TIMESTAMP, or

c) the YEAR of the patient's last visit to the medical center, if this is available in the row being migrated.

Basically, any desirable partitioning key value other than 0 may be used, though year-of-migration or year-of-last-visit may have desirable qualities, depending on the query application design. If the migration to slower, cheaper storage can be expressed using SQL predicates, such as a straightforward aging criteria against the timestamps which the customer's applications maintain in the patient rows already, a delete statement can be coded for Apply/MVS to execute automatically according to a schedule. Alternatively, application logic could be used to perform the delete according to more complicated procedural logic.

If the application would benefit from knowledge of the partitioning scheme, the partition_key column could be made visible. That way, an index on magnetic disk of (partition_key, patient_number) could support simple queries to determine the existence of "archived" records, to allow the end user the option of pursuing a query of all known records, or just those patient records in the current partition, referencing partition_key in their queries. WHERE PARTITION_KEY=0 would restrict the query to the current partition, with magnetic disk performance.

There are a number of factors to consider in this proposal:

1) Assuming that the indexes for all of the tables are maintained on magnetic disk, some care must be taken in writing the queries so as not to call up optical disk pages unnecessarily.

2) The partitioning key, an arbitrary column, becomes by definition the clustering key. Access paths for existing queries would likely be changed as a result of the change in clustering.

3) The indexes would become larger and suffer from having to index much more data.

As far as limiting the current data to a single DB2 4GB partition, the technique as outlined might be used over and over to "waterfall" current data from one partition to another, based on aging criteria. As we are discussing historical applications, if we can assume the presence of a timestamp column in the customer's data, then subscriptions could be defined to move data selectively from one partition to another based on age. The customer would choose how many of these partitions to back with magnetic disk and how many to back with optical disk. *The cost of repeatedly deleting and re-inserting the same row may be prohibitive, however.*

The deletion of rows from one partition, which triggers the "move" of a row from one partition to another, can be included as part of the subscription definition and can be made relative to CURRENT TIMESTAMP using a time/date calculation. For those applications where aging is the migration criteria, the process can potentially be automated using Apply/MVS definitions.

A few techniques should be considered to safeguard against unwanted optical disk reads:

1) Define subset views for applications needing only current data

2) Fetch just the primary keys and the partitioning keys first, and present the user with the knowledge that "archived" data exists for the answer set, allowing the user to choose whether to restrict the full query to current data only, or not. Hopefully, the key-fetching can be satisfied from an index. The customer may want to do this sort of test-the-water checking before unleashing a query against the optical drives.

When you’ve added new columns to a user table

From time to time, it is common practice to use the ALTER statement to add new columns to existing application tables. Just as view definitions insulate your applications from such underlying structure changes, registration and subscription definitions also are insulated from underlying changes in the user table. You continue to have available for copying only those columns selected during source table registration. What if you would like to have the new column appear in some or all of your copies?

First, you will need to have the column included in the registration. The easiest way to accomplish this is to *cancel* the existing registration, then register the source table anew. While easy to initiate, this results in the CD table being dropped and recreated, which may be an expensive process. In recreating the registration, CD_OLDEST_CHANGE is set to a higher value in the new registration. The Apply/x programs will analyze the situation and decide that there is a potential ’gap’ in the change data series. A wise assumption, as many thousands of CD table rows might have gone away with the dropping of the old CD table. This ’gap’ detection results in a cascading recovery phenomenon, whereby the Apply/x programs will attempt to full-refresh their complete copies. You may wish to avoid this type of automatic recovery if your copy tables are large. The following paragraphs will explain the necessary steps and considerations.

The DataPropagator Relational Guide states that Capture/MVS should be stopped normally before you begin altering registered source tables. This is not entirely correct. You may leave Capture/MVS running continuously, stop it gracefully, or simply re-initialize the Capture/MVS column mappings using the "re-init" command. You may choose to alter your CD and copy tables in place.

You must ALTER the CD table to add a new column to capture updates to the new column. Following that, re-initialize (or warm start) Capture/MVS. It should pick up the new column mappings during the re-initialization (or warm start).

As all the Apply/x implementations and DPropR/2 use dynamic SQL to access the BASE_TABLE, CD and CCD tables, the packages for these programs do not require rebind because of the ALTER. You can now proceed to adding the additional columns to some or all of your subscriptions.

We’ll build on the example scenario in the "Using DPRTOOLS in a data warehouse view scenario" section, and assume that a new column, ZIPCODE, has been added to the CENTRAL.CUSTOMER table and to the change data table associated with CENTRAL.CUSTOMER. The new column still needs to be added to the registered join-view and the copy table subscription. While it is possible to use the DB2 ALTER command to add the new ZIPCODE column to the copy table WAREHOUS.COPYJOIN4, the join views that comprise the view registration will need to be dropped and re-created, as there is no alter-add command for views.

```
--* The following views are registered as either base tables or change
--* data tables, and are dependents of the registered table
--* CENTRAL.CUSTOMER
--      view    Creator.Name
--    ------------------ ------------------
--   1          WAREHOUS.VIEWJOIN4
--   2          WAREHOUS.VIEWJOIN4A
--   3          WAREHOUS.VIEWJOIN4B
--   4          WAREHOUS.VIEWJOIN4C
--   5          WAREHOUS.VIEWJOIN4D
--   6          WAREHOUS.VIEWJOIN4E
--   7          WAREHOUS.VIEWJOIN4F
--   8          WAREHOUS.VIEWJOIN4G

-- re-create the view WAREHOUS.VIEWJOIN4
DROP VIEW WAREHOUS.VIEWJOIN4;

CREATE VIEW WAREHOUS.VIEWJOIN4 AS SELECT A.CUSTOMER_ID,A.BRANCH_NUMBER,
NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM CENTRAL.BRANCH A,
CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE D WHERE
A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4A
DROP VIEW WAREHOUS.VIEWJOIN4A;

CREATE VIEW WAREHOUS.VIEWJOIN4A AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4B
DROP VIEW WAREHOUS.VIEWJOIN4B;

CREATE VIEW WAREHOUS.VIEWJOIN4B AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM CENTRAL.BRANCH A,
CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.CD0000000000000002 D
WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4C
DROP VIEW WAREHOUS.VIEWJOIN4C;

CREATE VIEW WAREHOUS.VIEWJOIN4C AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4D
DROP VIEW WAREHOUS.VIEWJOIN4D;
```

```
CREATE VIEW WAREHOUS.VIEWJOIN4D AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM
CENTRAL.CD0000000000000003 A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C,
CENTRAL.BALANCE D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND
A.CUSTOMER_ID=C.CUSTOMER_ID AND C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4E
DROP VIEW WAREHOUS.VIEWJOIN4E;

CREATE VIEW WAREHOUS.VIEWJOIN4E AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4F
DROP VIEW WAREHOUS.VIEWJOIN4F;

CREATE VIEW WAREHOUS.VIEWJOIN4F AS SELECT IBMSNAP_UOWID,
IBMSNAP_INTENTSEQ,IBMSNAP_OPERATION,A.CUSTOMER_ID,A.BRANCH_NUMBER,NAME,
ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM CENTRAL.BRANCH A,
CENTRAL.CUSTOMER B,CENTRAL.CD0000000000000001 C, CENTRAL.BALANCE D
WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID AND
C.ACCT_ID=D.ACCT_ID;

-- re-create the view WAREHOUS.VIEWJOIN4G
DROP VIEW WAREHOUS.VIEWJOIN4G;

CREATE VIEW WAREHOUS.VIEWJOIN4G AS SELECT A.CUSTOMER_ID,
A.BRANCH_NUMBER,NAME,ADDRESS,C.ACCT_ID,ACCT_BALANCE,B.ZIPCODE FROM
CENTRAL.BRANCH A,CENTRAL.CUSTOMER B,CENTRAL.ACCOUNT C, CENTRAL.BALANCE
D WHERE A.CUSTOMER_ID=B.CUSTOMER_ID AND A.CUSTOMER_ID=C.CUSTOMER_ID
AND C.ACCT_ID=D.ACCT_ID;

--* commit work at DPROPRDB
--*
COMMIT;
```

And now that the registrations are assumed to be in order, it’s time to modify the subscription to include the new column as well:

```
-- The following subscriptions to CENTRAL.CUSTOMER will be altered:
--     user ID  cntl_svr copy_svr copy_table
--     -------- -------- -------- -------------------------------------
--   1 USERID   DPROPRDB DPROPRDB           WAREHOUS.COPYJOIN4

CONNECT TO DPROPRDB;
-- alter subscription #1
-- alter the target table
ALTER TABLE WAREHOUS.COPYJOIN4 ADD ZIPCODE CHAR(9);

-- create a new row in IBMSNAP_REF_COLS for added column 1
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',100,'A','ZIPCODE','N',7,'ZIPCODE');

-- create a new row in IBMSNAP_REF_COLS for added column 1
```

```
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',101,'A','ZIPCODE','N',7,'ZIPCODE');

-- create a new row in IBMSNAP_REF_COLS for added column 1
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',102,'A','ZIPCODE','N',7,'ZIPCODE');

-- create a new row in IBMSNAP_REF_COLS for added column 1
INSERT INTO USERID.IBMSNAP_REF_COLS(COPY_SERVER,COPY_OWNER,COPY_TABLE,
STMT_NUMBER,COL_TYPE,TARGET_NAME,IS_KEY,COLNO,EXPRESSION) VALUES(
'DPROPRDB','WAREHOUS','COPYJOIN4',103,'A','ZIPCODE','N',7,'ZIPCODE');

--* commit work at DPROPRDB
--*
COMMIT;
```

Large load using DSN1COPY

The following JCL and C language program provided and tested by Pat Shimer. This program is for customers wishing to use the DB2 DSN1COPY utility to load large DB2 copies which are exact 'clones' of the DB2 source tables.

```
//LGELD    JOB PHASE1,USER=USRT007,PASSWORD=USRT007
/*ROUTE PRINT STLVM1.SHIMER
//*********************************************
//*                                           *
//* LOADING LARGE TABLES - AUTOMAGICALLY      *
//*                                           *
//* PLEASE UPDATE JCL FOR YOUR ENVIRONMENT    *
//*                                           *
//*              EXAMPLE:                     *
//*                                           *
//*    1. DATA_SERVER = RDBD2041              *
//*    2. CNTL_SERVER = RDBD2041              *
//*    3. COPY_SERVER = RDBD2041              *
//*    4. COPY_OWNER  = USRT007               *
//*    5. COPY_TABLE  = COPY                  *
//*       SIMPLE TABLESPACE                   *
//*    6. BASE_OWNER  = USRT007               *
//*    7. BASE_TABLE  = BASE                  *
//*       SIMPLE TABLESPACE                   *
//*    8. COPY_STRUCTURE = 4                  *
//*    9. SUBSYSTEM IS DB2 2.3 (NAME=DB23) *
//*    10.PRUNING CONTROL TABLE NAME IS       *
//*       ASN.PRUNING_CONTROL.                *
//*    11.DATASET NAME FOR BASE TABLESPACE:*
//*  NAMING CONVENTION IS:                    *
//*CATALOG.DSNDBD.DATABASE.TABLESPACE.ID      *
//*                                           *
//*DSNC230.DSNDBD.DBCS3503.TLCS3508.I0001.A001*
```

```
//*       12.DATASET NAME FOR COPY TABLESPACE:*
//*DSNC230.DSNDBD.DBCS3503.TLCS3510.I0001.A001*
//*                                    *
//* ASSUMPTION: APPLY/MVS IS NOT RUNNING      *
//*   FOR THIS USERID                  *
//*   CAPTURE/MVS HAS BEEN STARTED PRIOR      *
//*   TO THIS JOB                      *
//*******************************************
//*******************************************
//*                                    *
//*  ZERO PRUNE LIMIT SEQUENCE AND SET        *
//*  LOGMARKER TIMESTAMP IN PRUNING           *
//*  CONTROL TABLE TO CAUSE CAPTURE/MVS TO    *
//*  START CAPTURE AND UPDATE REF_CNTL TO     *
//*  MATCH THE PRUNING CONTROL INFORMATION.   *
//*                                    *
//*******************************************
//DSNCT   EXEC PROC=TEP2PROC IN USER.PRIVATE.PROCLIB ON VOLUME USER01
//STEPLIB DD DISP=SHR,DSN=DB2.V2R3.DSNLOAD
//        DD  DSN=PLI.V2R3M0.SIBMLINK,DISP=SHR
//        DD  DSN=EDC.V2R1M0.SEDCLINK,DISP=SHR
//        DD DISP=SHR,DSN=DB223C.RUNLIB.LOAD
//TEP23.SYSIN DD *
*********************************************************************
* INSERT INTO PRUNING CONTROL                                       *
* THIS NEEDS TO BE AN INSERT IF CAPTURE/MVS WAS STARTED COLD SINCE    *
* CAPTURE/MVS CLEANS UP PRUNING CONTROL DURING COLD START             *
* FOR WARM START IT WILL BE AN UPDATE INSTEAD, E.G.                 *
* UPDATE ASN.PRUNING_CONTROL                                        *
*  IBMSNAP_LOGMARKER = CURRENT TIMESTAMP,                           *
*  PRUNE_LIMIT_SEQ = X'00000000000000000000'                        *
*  WHERE CNTL_SERVER = 'RDBD2041' AND                               *
*  COPY_SERVER = 'RDBD2041' AND                                     *
*  COPY_OWNER = 'USRT007' AND COPY_TABLE = 'COPY' AND               *
*  COPY_STRUCTURE = 4;                                              *
*********************************************************************
INSERT INTO ASN.PRUNING_CONTROL
  (IBMSNAP_LOGMARKER,PRUNE_LIMIT_SEQ,CNTL_SERVER,COPY_SERVER,
   COPY_OWNER,COPY_TABLE,COPY_STRUCTURE)
  VALUES(CURRENT TIMESTAMP,
  X'00000000000000000000',
  'RDBD2041',
  'RDBD2041',
  'USRT007',
  'COPY',
  4);
COMMIT;
*
  UPDATE IBMSNAP_REF_CNTL SET
    IBMSNAP_LOGMARKER = CURRENT TIMESTAMP;
COMMIT;
*
  UPDATE IBMSNAP_REF_CNTL SET
```

```
    LAST_DIFF_SEQUENCE = NULL;
COMMIT;
*
********************************************
*                                          *
*  DROP EXISTING COPY_TABLE AND RECREATE    *
*  LIKE BASE_TABLE. THIS IS TO GET RID OF   *
*  OVERHEAD COLUMNS - DSN1COPY WONT TOLERATE*
********************************************
DROP TABLE COPY;
COMMIT;
*
CREATE TABLESPACE TLCS3510 IN DBCS3503
  USING STOGROUP SYSDEFLT;
COMMIT;
CREATE TABLE COPY LIKE BASE
     IN DBCS3503.TLCS3510;
COMMIT;
*
/*
//********************************************
//*                                          *
//*  FOR BASE TABLE AND COPY TABLE:          *
//*  START DATABASE TO PUT TABLESPACE IN RO   *
//*  QUIESCE TABLESPACE AND STOP DATABASE     *
//*  SPACENAM                                *
//*  FOR DSN1COPY       DATASETS SYSUT1 AND 2 *
//*  PRIOR TO DSN1COPY                       *
//********************************************
//STRTSP   EXEC PGM=IKJEFT01
//* SPILL FILE WILL ACCEPT 10 COLUMNS OF SQL DATA - 10*280
//SPILL    DD  DSN=&&TEMP,DISP=(NEW,DELETE,DELETE),UNIT=SYSDA,
//             DCB=(DSORG=PS,LRECL=2800,RECFM=F,BLKSIZE=2800),
//             SPACE=(CYL,(5,5))
//STEPLIB  DD DSN=USER.RESLIB,DISP=SHR
//         DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//DBRMLIB  DD DSN=USRT007.OBJLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD *
 DSN S(DB23)
 -START DATABASE(DBCS3503) SPACENAM(TLCS3508) ACCESS(RO)
 -START DATABASE(DBCS3503) SPACENAM(TLCS3510) ACCESS(RO)
 END
/*
//QUIESCE EXEC DB23PROC,SYSTEM=DB23,UID='TEMP',UTPROC=''
//*
//********************************************
//*
//*  GENERATING JCL FOR THE QUIESCE UTILITY
//*
//********************************************
```

```
//*
//DB23PROC.SYSIN     DD  *
QUIESCE TABLESPACE DBCS3503.TLCS3508
QUIESCE TABLESPACE DBCS3503.TLCS3510
/*
//**********************************************
//*
//*  STOP TABLESPACES
//*
//**********************************************
//STOPSP    EXEC PGM=IKJEFT01
//* SPILL FILE WILL ACCEPT 10 COLUMNS OF SQL DATA - 10*280
//SPILL     DD  DSN=&&TEMP,DISP=(NEW,DELETE,DELETE),UNIT=SYSDA,
//              DCB=(DSORG=PS,LRECL=2800,RECFM=F,BLKSIZE=2800),
//              SPACE=(CYL,(5,5))
//STEPLIB   DD DSN=USER.RESLIB,DISP=SHR
//          DD DSN=ASNL.DEV423.LINKLIB,DISP=SHR
//          DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//          DD DSN=EDC.V2R1M0.SEDCLINK,DISP=SHR
//          DD DSN=PLI.V2R3M0.SIBMLINK,DISP=SHR
//DBRMLIB   DD DSN=USRT007.OBJLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD *
 DSN S(DB23)
 -STOP DATABASE(DBCS3503) SPACENAM(TLCS3508)
 -STOP DATABASE(DBCS3503) SPACENAM(TLCS3510)
 END
/*
//*********************************************
//* PRE-COMPILE                                 *
//*       ASNAQRY  TO GET DBID, OBID INFO       *
//*        FOR BASE TABLE AND COPY TABLE        *
//*********************************************
//PC       EXEC PGM=DSNHPC,PARM='HOST(C),SOURCE',REGION=5M
//STEPLIB   DD DSN=USER.RESLIB,DISP=SHR
//          DD DSN=DB2.V3R1.SDSNLOAD,DISP=SHR
//DBRMLIB  DD  DISP=SHR,DSN=USRT007.OBJLIB(ASNQDB)
//SYSPRINT DD  SYSOUT=*
//SYSTERM  DD  SYSOUT=*
//SYSUDUMP DD  DUMMY
//SYSCIN    DD  DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(800,(500,500))
//SYSUT1    DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2    DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSIN     DD  *
 /* **************** FIND SYSXLAT IDENTIFIER PROGRAM********* */
 #include "stdio.h"
 #include "limits.h"
 #include "string.h"
 #include "ctype.h"
 #include "time.h"
```

```
 #include <errno.h>
#define ok 0
/*******************************************************************/
/* Include SQL structures.                                         */
/*******************************************************************/
 EXEC SQL INCLUDE SQLCA;
 EXEC SQL BEGIN DECLARE SECTION;
/*******************************************************************/
/*                                                                 */
     char tsnam1 ??(9??);  /* SYSIBM.SYSTABLES TSNAME      */
     char tsnam2 ??(9??);  /* SYSIBM.SYSTABLES TSNAME      */
     short obid1,obid2;        /* SYSIBM.SYSTABLES OBID        */
     short psid1,psid2;        /* SYSIBM.SYSTABLESPACE PSID   */
     short dbid1,dbid2;        /* SYSIBM.SYSTABLESPACE DBID   */


 EXEC SQL END DECLARE SECTION;
 char  *blankptr;                  /* first trailing blank ptr    */
 FILE *fp2;    /* FILE type for stream functions */
/* Output buffer                                                   */
 char outbuf??(81??);
 char  inbuf??(81??);
 char blnk = ' ';

/*******************************************************************/
/*  GLBL  VARIABLES                                                */
/*******************************************************************/
 short j,k;
main()
 {
   short ii,jj,kk;
   char strbuf ??(80??);
  /*******************************************************************/
  /* Find DBID and OBID of source table and target table            */
  /*******************************************************************/
/* Open output                                                     */
/*******************************************************************/
/* Format of       OUTPUT DATASET - SEQUENTIAL DATASET fwrite      */
/*          (QSAM), FIXED RECORDS , BLOCKSIZE 80                   */
/* DDNAME IS OUTDD IN JCL                                          */
/*******************************************************************/
/*OUTDD    DD DSN=&&XLATIN,DISP=(NEW,PASS),UNIT=SYSDA,     */
/*          SPACE=(50,(50,50)),RECFM=FB, LRECL=80,        */
/*          BLKSIZE=800                                    */
  fp2   = fopen("dd:OUTDD",
    "wb,type=record,recfm=FB,lrecl=80,blksize=800");

  if (fp2 == NULL)
    {
      perror(" fopen error on OUTDD");
      printf(" fopen error on OUTDD errno %d",errno);
      exit(10);
    }
```

```
/*      Initialize output buffer                                   */
    outbuf??(81??) = '\0';
  /* For this example the base  table has CREATOR = USRT007 and      */
  /* NAME= BASE. Put your base table information here.               */
    EXEC SQL
      SELECT TSNAME INTO :tsnam1 FROM SYSIBM.SYSTABLES WHERE
      CREATOR = 'USRT007'
      AND NAME= 'BASE';
  if (SQLCODE == ok)
    {
      /* find the 1st trail. blnk*/
      blankptr = strchr(tsnam1,' ');
      /* truncate trailing blanks   */
      if (blankptr != NULL)
       *blankptr = '\0';
      /* For this example the base  table has CREATOR = USRT007 and */
      /* NAME= COPY. Put your copy table information here.          */
      EXEC SQL
       SELECT TSNAME INTO :tsnam2 FROM SYSIBM.SYSTABLES WHERE
       CREATOR = 'USRT007'
       AND NAME= 'COPY';
      if (SQLCODE == ok)
        {
         /* find the 1st trail. blnk*/
         blankptr = strchr(tsnam2,' ');
         /* truncate trailing blanks   */
         if (blankptr != NULL)
           *blankptr = '\0';
         EXEC SQL
           SELECT DBID,PSID INTO :dbid1,psid1 FROM
           SYSIBM.SYSTABLESPACE WHERE NAME = :tsnam1;
         if (SQLCODE == ok)
           {
             EXEC SQL
             SELECT DBID,PSID INTO :dbid2,psid2 FROM
             SYSIBM.SYSTABLESPACE WHERE NAME = :tsnam2;
             if (SQLCODE == ok)
               {
  /* For this example the base  table has CREATOR = USRT007 and      */
  /* NAME= BASE. Put your base table information here.               */
                 EXEC SQL
                 SELECT OBID INTO :obid1 FROM SYSIBM.SYSTABLES WHERE
                 TSNAME = :tsnam1 AND
                 CREATOR = 'USRT007'
                 AND NAME= 'BASE';
                 if (SQLCODE == ok)
                   {
      /* For this example the base  table has CREATOR = USRT007 and */
      /* NAME= COPY. Put your copy table information here.          */
                     EXEC SQL SELECT OBID
                     INTO :obid2 FROM SYSIBM.SYSTABLES WHERE
                     TSNAME = :tsnam2 AND
                     CREATOR = 'USRT007'
```

```
                AND NAME= 'COPY';
                if (SQLCODE == ok)
                  {
                /************************************ */
                /* Output SYSXLAT file 3 records:      */
                /* dbid1,dbid2 (decimal)               */
                /* psid1,psid2 (decimal)               */
                /* obid1,obid2 (decimal)               */
                /************************************ */

                    sprintf(strbuf,"%d , %d",dbid1,dbid2);
                    printf("%s",strbuf);
                    strcpy(outbuf,strbuf);
                    /* write output record              */
                    j =fwrite(outbuf,1, 80, fp2);
                    if (j != 80)
                      exit(20);
                    sprintf(strbuf,"%d , %d",psid1,psid2);
                    printf("%s",strbuf);
                    strcpy(outbuf,strbuf);
                    j =fwrite(outbuf,1, 80, fp2);
                    if (j != 80)
                      exit(30);
                    sprintf(strbuf,"%d , %d",obid1,obid2);
                    printf("%s",strbuf);
                    strcpy(outbuf,strbuf);
                    j =fwrite(outbuf,1, 80, fp2);
                    if (j != 80)
                      exit(40);
                /* Close  SYSXLAT file 3 records:      */
/*************************************************************/
/* EOF FILE PROCESSING. CLOSE          OUTPUT FILE           */
/*************************************************************/
                    if (fclose(fp2) != 0)
                      {
                        printf("fclose of OUTDD failed");
                        printf(" fclose errno %d",errno);
                        exit(50);
                      }
                }
          else exit(90);
              }
        else exit(80);
            }
      else exit(70);
          }
    else exit(60);
      }
  else exit(50);
   }
 else exit(40);

 }
```

```
/*
//******************************************
//*      COMPILE                            *
//*        ASNQRY  TO GET DBID, OBID INFO   *
//*        FOR BASE TABLE AND COPY TABLE    *
//******************************************
//C        EXEC PGM=EDCCOMP,COND=(4,LT,PC),REGION=4096K,
//             PARM=('OPTIMIZE')
//STEPLIB  DD  DSN=EDC.V2R1M0.SEDCLINK,DISP=SHR
//         DD  DSN=PLI.V2R3M0.SIBMLINK,DISP=SHR
//         DD  DSN=EDC.V2R1M0.SEDCCOMP,DISP=SHR
//SYSMSGS  DD  DSN=EDC.V2R1M0.SEDCMSGS(EDCMSGE),DISP=SHR
//SYSLIB   DD  DSN=EDC.V2R1M0.SEDCHDRS,DISP=SHR
//SYSLIN   DD  DISP=SHR,DSN=USRT007.OBJLIB(ASNQRY)
//SYSPRINT DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSTERM  DD  DUMMY
//SYSIN    DD  DSN=&&DSNHOUT,DISP=(OLD,DELETE)
//SYSUT1   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT2   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT3   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT6   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9   DD  UNIT=SYSDA,DISP=(NEW,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  DUMMY
//******************************************
//* LINK-EDIT                               *
//*        ASNQRY  TO GET DBID, OBID INFO   *
//*        FOR BASE TABLE AND COPY TABLE    *
//******************************************
//*            COMPRESS
//COMPRESS EXEC PGM=IEBCOPY
```

```
//SYSPRINT DD SYSOUT=*
//A          DD DISP=SHR,DSN=ASNL.DEV423.LINKLIB
//SYSIN     DD *
  C I=A,O=A
/*
//*              PRE-LINKEDIT
//LKED      EXEC PGM=IEWL,PARM='MAP,AMODE=31,RMODE=ANY'
//SYSLIB     DD DSN=EDC.V2R1M0.SEDCBASE,DISP=SHR
//           DD DSN=PLI.V2R3M0.SIBMBASE,DISP=SHR
//           DD DSN=USER.RESLIB,DISP=SHR
//           DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//*         NETVIEW LIBRARIES
//           DD DSN=SYS1.CNMLINK,DISP=SHR
//           DD DSN=SYS1.CNMINST,DISP=SHR
//           DD DSN=SYS1.CNMSAMP,DISP=SHR
//           DD DSN=SYS1.BNJMISC,DISP=SHR
//           DD DSN=SYS1.BNJPNL1,DISP=SHR
//           DD DSN=SYS1.BNJPNL2,DISP=SHR
//OBJ        DD DSN=USRT007.OBJLIB,DISP=SHR
//SYSLMOD    DD DSN=ASNL.DEV423.LINKLIB(ASNQRY),DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSUT1     DD DSN=&&SYSUT1,UNIT=SYSDA,DISP=(NEW,DELETE),
//     SPACE=(32000,(30,30))
//SYSLIN     DD DDNAME=SYSIN
//SYSIN      DD *
  INCLUDE OBJ(ASNQRY)
  NAME ASNQRY(R)
/*
//*********************************************
//*                                            *
//* BIND   ASNQRY   TO GET DBID, OBID INFO       *
//*        FOR BASE TABLE AND COPY TABLE        *
//*********************************************
//BIND      EXEC PGM=IKJEFT01
//STEPLIB   DD DSN=USER.RESLIB,DISP=SHR
//          DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//DBRMLIB  DD DSN=USRT007.OBJLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD *
 DSN S(DB23)
 BIND PLAN(ASNQPLAN) MEMBER(ASNQDB) ACTION(REP) ACQUIRE(USE)
 END
/*
```

```
//********************************************
//*                                  *
//*  RUN ASNQRY  TO GET DBID, OBID INFO       *
//*       FOR BASE TABLE AND COPY TABLE        *
//********************************************
//QRY       EXEC PGM=IKJEFT01
//* SPILL FILE WILL ACCEPT 10 COLUMNS OF SQL DATA - 10*280
//SPILL     DD  DSN=&&TEMP,DISP=(NEW,DELETE,DELETE),UNIT=SYSDA,
//              DCB=(DSORG=PS,LRECL=2800,RECFM=F,BLKSIZE=2800),
//              SPACE=(CYL,(5,5))
//STEPLIB   DD DSN=USER.RESLIB,DISP=SHR
//          DD DSN=ASNL.DEV423.LINKLIB,DISP=SHR
//          DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//          DD DSN=EDC.V2R1M0.SEDCLINK,DISP=SHR
//          DD DSN=PLI.V2R3M0.SIBMLINK,DISP=SHR
//DBRMLIB  DD DSN=USRT007.OBJLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD *
 DSN S(DB23)
 RUN  PROGRAM(ASNQRY) PLAN(ASNQPLAN) -
      LIB('ASNL.DEV423.LINKLIB')
 END
/*
//*OUTDD    DD  DSN=USRT007.XLATIN,DISP=SHR
//OUTDD    DD  DSN=&&XLATIN,DISP=(NEW,PASS),UNIT=SYSDA,
//              DCB=(DSORG=PS,LRECL=80,RECFM=FB,BLKSIZE=800),
//              SPACE=(CYL,(5,5))
//********************************************
//*                                  *
//*  RUN DSN1COPY TO LOAD COPY TABLE FROM      *
//*       BASE TABLE.                  *
//********************************************
//DSN1CPY EXEC PGM=DSN1COPY,PARM='OBIDXLAT,RESET'
//STEPLIB  DD DSN=DSN230.DSNLOAD,DISP=SHR
//SYSPRINT DD  SYSOUT=A
//* DATASET NAME OF BASE TABLE TABLESPACE
//* MVS/DFP
//* NAMING CONVENTION IS CATALOG.DSNDBD.DATABASE.TABLESPACE.I0001.A001
//* FOR EACH SEGMENT
//SYSUT1   DD  DSN=DSNC230.DSNDBD.DBCS3503.TLCS3508.I0001.A001,
//              DISP=OLD
//* DATASET NAME OF COPY TABLE TABLESPACE
//SYSUT2   DD  DSN=DSNC230.DSNDBD.DBCS3503.TLCS3510.I0001.A001,
//              DISP=OLD
//*SYSXLAT  DD  DSN=USRT007.XLATIN,DISP=SHR,UNIT=SYSDA,VOL=SER=USER01
//SYSXLAT  DD  DSN=&&XLATIN,DISP=(OLD,DELETE)
//*SYSXLAT  DD  *
//*263,4
//*2,293
//*3,321
//*
```

```
//*********************************************
//*                                          *
//*  ADD OVERHEAD COLUMN  BACK TO COPY TABLE  *
//*  ADD BEFORE-IMAGE COLUMN  TO COPY TABLE   *
//*********************************************
//ALTCOL  EXEC PROC=TEP2PROC IN USER.PRIVATE.PROCLIB ON VOLUME USER01
//STEPLIB DD DISP=SHR,DSN=DB2.V2R3.DSNLOAD
//        DD  DSN=PLI.V2R3M0.SIBMLINK,DISP=SHR
//        DD  DSN=EDC.V2R1M0.SEDCLINK,DISP=SHR
//        DD DISP=SHR,DSN=DB223C.RUNLIB.LOAD
//TEP23.SYSIN DD *
        ALTER TABLE COPY ADD
          IBMSNAP_LOGMARKER TIMESTAMP;
COMMIT;
        ALTER TABLE COPY ADD
          XTEST_DATA INTEGER;
COMMIT;
/*
//*********************************************
//*                                          *
//*  FOR BASE TABLE AND COPY TABLE:           *
//*  START DATABASE TO START TABLESPACE BACK  *
//*  UP                                      *
//*********************************************
//STRTSP   EXEC PGM=IKJEFT01
//* SPILL FILE WILL ACCEPT 10 COLUMNS OF SQL DATA - 10*280
//SPILL     DD  DSN=&&TEMP,DISP=(NEW,DELETE,DELETE),UNIT=SYSDA,
//              DCB=(DSORG=PS,LRECL=2800,RECFM=F,BLKSIZE=2800),
//              SPACE=(CYL,(5,5))
//STEPLIB  DD DSN=USER.RESLIB,DISP=SHR
//         DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//DBRMLIB  DD DSN=USRT007.OBJLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD *
 DSN S(DB23)
 -START DATABASE(DBCS3503) SPACENAM(TLCS3508)
 -START DATABASE(DBCS3503) SPACENAM(TLCS3510)
 END
/*
```

```
//*******************************************
//*                                         *
//*  START APPLY/MVS                        *
//*******************************************
//EXECUTE EXEC PGM=ASNAP,PARM='DB23 M S',REGION=3M
//STEPLIB DD    DISP=SHR,DSN=ASNL.LINKLIB,
//              UNIT=SYSDA,VOL=SER=RMS002
//         DD DSN=USER.RESLIB,DISP=SHR
//         DD DSN=DB2.V2R3.DSNLOAD,DISP=SHR
//         DD DSN=CEE.V1R2M0.SCEERUN,DISP=SHR
//*        DD DSN=PLI.V2R3M0.SIBMLINK,DISP=SHR
//MSGS      DD DSN=USRT007.ASNA.EMSGS,DISP=SHR
//SYSTERM  DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//DSNTRACE DD DUMMY
//SYSPRINT DD SYSOUT=*
```

Table Formats

Base aggregate and change aggregate:

```
CREATE TABLE agg_copy(
  IBMSNAP_LLOGMARKER TIMESTAMP,
  IBMSNAP_HLOGMARKER TIMESTAMP,
  <calculated columns> <,group-by columns>);
```

Change Data table:

```
CREATE TABLE cd (
IBMSNAP_UOWID CHAR(10) FOR BIT DATA  NOT NULL,
IBMSNAP_INTENTSEQ CHAR(10) FOR BIT DATA  NOT NULL,
IBMSNAP_OPERATION CHAR(1) NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP NOT NULL,
<after-change columns> <,before-change columns> );
```

Consistent Change Data table:

```
CREATE TABLE CCD (
IBMSNAP_INTENTSEQ CHAR(10) FOR BIT DATA  NOT NULL,
IBMSNAP_OPERATION CHAR(1) NOT NULL,
IBMSNAP_COMMITSEQ CHAR(10) FOR BIT DATA  NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP NOT NULL,
<after-change columns> <,before-change colums> );
```

Unit of Work table:

```
CREATE TABLE ASN.IBMSNAP_UOW (
IBMSNAP_UOWID CHAR(10) FOR BIT DATA  NOT NULL,
IBMSNAP_COMMITSEQ CHAR(10) FOR BIT DATA  NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP NOT NULL,
IBMSNAP_AUTHTKN CHAR(12) NOT NULL,
IBMSNAP_AUTHID CHAR(18) NOT NULL);
```

Change Data Control, or *Registration* table:

```
CREATE TABLE ASN.IBMSNAP_CD_CNTL (
BASE_OWNER CHAR(18) NOT NULL,
BASE_TABLE CHAR(18) NOT NULL,
CRIT_SECTION_OWNER CHAR(18) NOT NULL,
CRIT_SECTION_TABLE CHAR(18) NOT NULL,
CD_OWNER CHAR(18),
CD_TABLE CHAR(18),
PRUNE_CNTL_OWNER CHAR(18) NOT NULL
PRUNE_CNTL_TABLE CHAR(18) NOT NULL,
CD_OLDEST_CHANGE CHAR(10) FOR BIT DATA ,
FULLREFRESHENABLE SMALLINT NOT NULL,
CCD_OWNER CHAR(18),
CCD_TABLE CHAR(18),
CCD_OLDEST_CHANGE CHAR(10) FOR BIT DATA,
CD_CONDENSED CHAR(1) ,
CD_CONSISTENT CHAR(1) ,
CD_COMPLETE   CHAR(1) ,
CCD_CONDENSED CHAR(1) ,
CCD_CONSISTENT CHAR(1) ,
CCD_COMPLETE   CHAR(1) ,
BASE_STRUCTURE SMALLINT NOT NULL,
BASE_CONDENSED CHAR(1) NOT NULL,
BASE_CONSISTENT CHAR(1) NOT NULL,
BASE_COMPLETE   CHAR(1) NOT NULL,
REGISTRAR CHAR(18) NOT NULL,
ARCH_LEVEL CHAR(4) NOT NULL,
DESCRIPTION CHAR(254),
PREFIX_CHAR VARCHAR(4));
```

Pruning control table:

```
CREATE TABLE pruning_cntl (
COPY_SERVER CHAR (18) NOT NULL,
COPY_OWNER CHAR (18) NOT NULL,
COPY_TABLE CHAR(18) NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,
PRUNE_LIMIT_SEQ CHAR(10) FOR BIT DATA,
CNTL_SERVER CHAR(18) NOT NULL,
COPY_STRUCTURE SMALLINT NOT NULL);
```

Refresh control or *subscription* table:

```
CREATE TABLE
IBMSNAP_REF_CNTL (
ENABLE SMALLINT NOT NULL,
DATA_SERVER CHAR(18) NOT NULL,
BASE_OWNER CHAR (18) NOT NULL,
BASE_TABLE CHAR(18) NOT NULL,
STMT_NUMBER SMALLINT NOT NULL,
PRIORITY SMALLINT NOT NULL,
COPY_OWNER CHAR (18) NOT NULL,
COPY_TABLE CHAR(18) NOT NULL,
LAST_DIFF_SEQUENCE CHAR(10) FOR BIT DATA,
RUNMODE CHAR(2) NOT NULL,
STATUS SMALLINT NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,
LASTRUN TIMESTAMP,
INTERVAL_MINUTES INT NOT NULL,
COPY_CONDENSED CHAR(1) NOT NULL,
COPY_COMPLETE CHAR(1) NOT NULL,
COPY_CONSISTENT CHAR(1) NOT NULL,
COPY_STRUCTURE SMALLINT NOT NULL,
ARCH_LEVEL CHAR(4) NOT NULL,
COPY_SERVER CHAR(18) NOT NULL,
EXPIRES DATE,
PREDICATES VARCHAR (512),
SQL_STMT VARCHAR(1024));
```

Refresh columns or *subscription columns*:

```
CREATE TABLE IBMSNAP_REF_COLS(
COPY_SERVER CHAR(18) NOT NULL,
COPY_OWNER CHAR(18) NOT NULL,
COPY_TABLE CHAR(18) NOT NULL,
STMT_NUMBER SMALLINT NOT NULL,
COL_TYPE CHAR(1) NOT NULL,
TARGET_NAME CHAR(18) NOT NULL,
IS_KEY CHAR(1) NOT NULL,
COLNO SMALLINT NOT NULL,
EXPRESSION VARCHAR(254) NOT NULL);
```

Refresh audit trail table:

```
CREATE TABLE IBMSNAP_TRAIL (
ENABLE SMALLINT NOT NULL,
DATA_SERVER CHAR(18) NOT NULL,
BASE_OWNER CHAR (18) NOT NULL,
BASE_TABLE CHAR(18) NOT NULL,
STMT_NUMBER SMALLINT NOT NULL,
COPY_OWNER CHAR (18) NOT NULL,
COPY_TABLE CHAR(18) NOT NULL,
LAST_DIFF_SEQUENCE CHAR(10) FOR BIT DATA,
RUNMODE CHAR(2) NOT NULL,
STATUS SMALLINT NOT NULL,
IBMSNAP_LOGMARKER TIMESTAMP,
LASTRUN TIMESTAMP,
FETCHED       INT NOT NULL,
FULLREFRESHED INT NOT NULL,
INSERTED      INT NOT NULL,
DELETED       INT NOT NULL,
UPDATED       INT NOT NULL,
REWORKED      INT NOT NULL,
INTERVAL_MINUTES INT NOT NULL,
SQLSTATE CHAR(5),
SQLCODE INTEGER,
SQLERRP CHAR(8),
APPERRN CHAR(8),
COPY_SERVER CHAR(18) NOT NULL,
CNTL_SERVER CHAR(18) NOT NULL,
SQLERRM VARCHAR(70),
APPERRM VARCHAR(760));
```

Capture/MVS tuning parameters:

```
CREATE TABLE ASN.IBMSNAP_CCPPARMS (
RETENTION_LIMIT INT,
LAG_LIMIT INT,
COMMIT_INTERVAL INT);
```

Capture/MVS execution trace table:

```
CREATE TABLE ASN.IBMSNAP_TRACE
(OPERATION   CHAR(8)       NOT NULL,
TRACE_TIME   TIMESTAMP     NOT NULL,
DESCRIPTION  VARCHAR(254)  NOT NULL);
```

Subscriber profile table:

```
CREATE TABLE userid.IBMSNAP_PROFILE (
APPLYID CHAR(18) NOT NULL,
COPYSERVER  CHAR(18) NOT NULL,
CNTLSERV_F CHAR(1) NOT NULL,
PRIMARY KEY (APPLYID));
```

Authorization control table:

```
CREATE TABLE ASN.IBMSNAP_AUTH (
GRANTOR  CHAR(18) NOT NULL,
GRANTEE  CHAR(18) NOT NULL,
GRANTOR_HIGH CHAR(1) NOT NULL,
GRANTEE_LEVEL CHAR(1) NOT NULL,
GRANTEE_STATUS CHAR(1) NOT NULL);
```

Apply/x routing table for control server lookup:

```
CREATE TABLE IBMSNAP_ROUTING (
CNTL_SERVER CHAR(18) NOT NULL,
TRY_CONNECT_TIME TIMESTAMP);
```

Source and Target combinations supported by DPropR/2

	BASE_STRUCTURE	BASE_COMPLETE	BASE_CONSISTENT	BASE_CONDENSED	CD_TABLE	CD_COMPLETE	CD_CONSISTENT
1_	BA	N	T	A	?	?	?
2_	CA	N	T	A	?	?	?
3_	CCD	Y	T	N	?	?	?
4_	CCD	Y	T	N	?	?	?
5_	CCD	Y	T	N	?	?	?
6_	CCD	Y	T	N	?	?	?
7_	CCD	Y	T	N	?	?	?
8_	CCD	Y	T	N	?	?	?
9_	CCD	Y	T	P	?	?	?
10_	CCD	Y	T	P	?	?	?
11_	CCD	Y	T	P	?	?	?
12_	CCD	Y	T	Y	?	?	?
13_	CCD	Y	T	Y	?	?	?
14_	CCD	Y	T	Y	?	?	?
15_	CCD	Y	T	Y	?	?	?
16_	PIT	Y	C	Y	?	?	?
17_	PIT	Y	T	Y	?	?	?
18_	PIT	Y	T	Y	?	?	?
19_	USER	Y	T	Y	CD_TABLE	N	C
20_	USER	Y	T	Y	CD_TABLE	N	T
21_	USER	Y	T	Y	CD_TABLE	N	T
22_	USER	Y	T	Y	CD_TABLE	N	T
23_	USER	Y	T	Y	CD_TABLE	N	T
24_	USER	Y	T	Y	CD_TABLE	N	T
25_	USER	Y	T	Y	CD_TABLE	N	T
26_	USER	Y	T	Y	CD_TABLE	N	T
27_	USER	Y	T	Y	CD_TABLE	N	T
28_	USER	Y	T	Y	CD_TABLE	N	T
29_	USER	Y	T	Y	CD_TABLE	N	T
30_	USER	Y	T	Y	CD_TABLE	N	T
31_	USER	Y	T	Y	CD_TABLE	N	T
32_	USER	Y	T	Y	CD_TABLE	N	T
33_	USER	Y	T	Y	CD_TABLE	N	T
34_	USER	Y	T	Y	CD_TABLE	N	T
35_	USER	Y	T	Y	CD_TABLE	N	T
36_	USER	Y	T	Y	CD_TABLE	N	T
37_	USER	Y	T	Y	CD_TABLE	N	T
38_	USER	Y	T	Y	CD_TABLE	N	T
39_	USER	Y	T	Y	CD_TABLE	N	T
40_	USER	Y	T	Y	CD_TABLE	N	T
41_	USER	Y	T	Y	?	?	?
42_	USER	Y	T	Y	?	?	?

	CD_CONDENSED	CCD_TABLE	CCD_COMPLETE	CCD_CONSISTENT	CCD_CONDENSED	COPY_STRUCTURE	COPY_COMPLETE		COPY_CONSISTENT	COPY_CONDENSED
1_	?	?	?	?	?	BA	N	1_	T	A
2_	?	?	?	?	?	CA	N	2_	T	A
3_	?	?	?	?	?	CA	N	3_	T	A
4_	?	?	?	?	?	CCD	N	4_	T	N
5_	?	?	?	?	?	CCD	N	5_	T	Y
6_	?	?	?	?	?	CCD	Y	6_	T	N
7_	?	?	?	?	?	CCD	Y	7_	T	Y
8_	?	?	?	?	?	PIT	Y	8_	T	Y
9_	?	?	?	?	?	CCD	N	9_	T	Y
10_	?	?	?	?	?	CCD	Y	0_	T	Y
11_	?	?	?	?	?	PIT	Y	1_	T	Y
12_	?	?	?	?	?	CA	N	2_	T	A
13_	?	?	?	?	?	CCD	N	3_	T	Y
14_	?	?	?	?	?	CCD	Y	4_	T	Y
15_	?	?	?	?	?	PIT	Y	5_	T	Y
16_	?	?	?	?	?	PIT	Y	6_	C	Y
17_	?	?	?	?	?	BA	N	7_	T	A
18_	?	?	?	?	?	PIT	Y	8_	T	Y
19_	N	?	?	?	?	PIT	Y	9_	C	Y
20_	N	CCD_TABLE	N	T	N	BA	N	0_	T	A
21_	N	CCD_TABLE	N	T	N	CA	N	1_	T	A
22_	N	CCD_TABLE	N	T	N	CCD	N	2_	T	N
23_	N	CCD_TABLE	N	T	N	CCD	N	3_	T	Y
24_	N	CCD_TABLE	N	T	N	CCD	Y	4_	T	N
25_	N	CCD_TABLE	N	T	N	CCD	Y	5_	T	Y
26_	N	CCD_TABLE	N	T	N	PIT	Y	6_	T	Y
27_	N	CCD_TABLE	N	T	Y	BA	N	7_	T	A
28_	N	CCD_TABLE	N	T	Y	CA	N	8_	T	A
29_	N	CCD_TABLE	N	T	Y	CCD	N	9_	T	N
30_	N	CCD_TABLE	N	T	Y	CCD	N	0_	T	Y
31_	N	CCD_TABLE	N	T	Y	CCD	Y	1_	T	N
32_	N	CCD_TABLE	N	T	Y	CCD	Y	2_	T	Y
33_	N	CCD_TABLE	N	T	Y	PIT	Y	3_	T	Y
34_	N	?	?	?	?	BA	N	4_	T	A
35_	N	?	?	?	?	CA	N	5_	T	A
36_	N	?	?	?	?	CCD	N	6_	T	N
37_	N	?	?	?	?	CCD	N	7_	T	Y
38_	N	?	?	?	?	CCD	Y	8_	T	N
39_	N	?	?	?	?	CCD	Y	9_	T	Y
40_	N	?	?	?	?	PIT	Y	0_	T	Y
41_	?	?	?	?	?	BA	N	1_	T	A
42_	?	?	?	?	?	PIT	Y	2_	T	Y

Transactions which will not propagate correctly

There is at least one known case which is, as yet, impossible to propagate correctly. Assume that the base table includes an integer column, KEY_COL, which is used as the primary key in a point in time copy of the base table. Let's say that an application *updates* the KEY_COL column in the following manner:

```
UPDATE BASE SET KEY_COL = KEY_COL +1;
```

If you think through how this will propagate, you see that the Apply process will be modifying the primary key values as it is attempting to apply the updates, quickly losing its ability to properly locate rows in the copy.

References

ADIBA80 Michel E. Adiba, Bruce G. Lindsay, "Data Base Snapshots", IBM Research Report RJ2772, March 7, 1980

DATE87 C.J. Date, "Twelve rules for a distributed data base", Computerworld, June 8, 1987, also

C.J. Date, "What is a Distributed Database System?", InfoDB, Volume 2 Number 2 and Volume 2 Number 3, Summer and Fall 1987

DEC88 VAX Data Distributor Handbook, Digital Equipment Corp., July 1988, order number AA-HZ65B-TE

EDEL93 Herb Edelstein, "Replicating Data, *DBMS*, June, 1993

GOLD90 Rob Goldring, "Application Design for Distributed DB2", *Database Programming & Design*, September, 1990

GOLD93 Rob Goldring, "Introduction to Time Series Data", *InfoDB*, Volume 7 Number 2, Spring 1993

GOLD94 Rob Goldring, "A Discussion of Relational Database Replication Technology", *InfoDB*, Volume 8 Number 1, Spring 1994, IBM reprint order number GC26-8369-00

GOLD95 Rob Goldring, "Update Replication: What Every Designer Should Know", *InfoDB*, Volume 9 Number 2, Spring 1995, IBM reprint order number GC26-8459-00

HAAS86 Laura M Haas, Bruce G. Lindsay, "Method for refreshing multicolumn tables in a relational data base using minimal information", United States Patent number: 4,631,673; Date of Patent: December 23, 1986

IBM951 Don Cameron, editor, "Data Where You Need It, The DPropR Way! DataPropagator Relational Solutions Guide", IBM order number GG24-4492-00, June 1995, *an IBM red book*

IBM952 Julie Wood, "DataPropagator Relational Guide, Release 2", IBM order number SC26-3399-03, August 1995, *the official product manual for everything but DPropR/400 specific information*

WHITE92 Colin White, "Distributing and Accessing Corporate Data", InfoDB, Volume 6 Number 4, Summer 1992