



Implementing Journaled Files And Transaction Tracking In BBj 2.0

By John Schroeder and Chris Hardekopf

One of the most powerful additions to BBj® in Revision 2.0 is the Journaled File System, which allows you to safeguard your data and preserve its integrity. Journaled files and transactions allow applications to process a set of file operations without risking the loss of any of the set due to a system crash. A journaled system with transactions controls the process of grouping file operations so that either all operations (i.e. all the multiple disk writes required to update a record) are on disk or no operations are on disk.

The principal benefit of journaled files is their resistance to corruption. Because all operations on journaled files are atomic-operations either completely succeed or completely fail-no operation is ever half-completed, thereby corrupting the file.

Transactions build on this characteristic of journaled files, enabling the application programmer to define sets of operations to be treated as a single operation on one or more files. This set of operations will either completely succeed or fail in its entirety, in the case of a catastrophic system failure.

In this article we will examine journaled files and how to use them with transactions.

JOURNALED FILES

Creating a journaled file is similar to creating an MKEYED file. Use the JKEYED verb with the same syntax as the MKEYED verb. For example, you can use the following statement to create a journaled file with two keys, with a descending sort on the second key:

```
JKEYED "MYFILE",[1:1:2],[2:1:10:"D"],0,128
```

This specifies the creation of a JKEYED file with two keys, the first being the first two bytes of the first field in the record, while the second is the first 10 bytes of the second field, in descending sequence. The record size is 128 bytes per record.

You can also define a "single keyed" JKEYED file, similarly to a DIRECT file as in:

```
JKEYED "MYFILE",12,0,80
```

which defines a file with a single key of 12 bytes and a record size of 80 bytes. Note that all JKEYED files are dynamic. The record count parameter is required for the syntax, but ignored in creating the file.

Once a JKEYED file is defined, it must be opened with the JOPEN verb and erased with the JERASE verb. The normal, BEGIN, END, STOP, START RELEASE, or CLOSE verbs will close a JKEYED file. The JOPEN and JERASE verbs are required, since a journaled file is not a single file, but a data file and log file pair. In addition, the JOPEN can associate a journaled file with a transaction.

The normal READ, WRITE, and REMOVE verbs are used with journaled files in the same way as with other keyed files. They are different in that when a journaled file is updated i.e., with the REMOVE or WRITE verbs, the data file is updated, and in addition, a record is written to the log file so that a complete history of updates is maintained.

TRANSACTIONS

A transaction is defined as a group of updates to one or more files that are tied together so that all are completed, or none are. It is also called an "atomic" operation because the entire transaction is treated as a unit. Normally, a transaction is either explicitly committed or rolled back. There are cases where the commit or rollback is automatic, as we shall see. Building on journaled files, BBj can manage transactions.

Transactions are set up using the TOPEN function. This function returns a numeric value, or handle, that identifies the transaction. This handle is used to associate journaled files with the transactions and to identify the transaction in a commit or rollback operation.

The syntax for the TOPEN function is:

```
handle=TOPEN(int{,ERR=stmt})
```

where handle is a numeric variable, and int takes on the values 1, 2, or 3, depending on the level isolation that is required for the transaction. It determines how multiple concurrent transactions interact. These values are explained below:

VALUE	MEANING
1	This is the lowest level of isolation. It allows the operations in the transaction to read data even though the data may have been written out during a transaction that has not yet been committed or rolled back. The risk is that the data read will be changed after the read, because a rollback was executed. Hence the transaction could be incorrect.
2	The transaction will not be able to read data involved in another transaction until that update has been committed or rolled back. Here you have a higher level of isolation than in number 1, but after you read it, you are still open to another process-writing data that you have read.
3	No data read in the transaction can be changed by any other transaction until the transaction has been committed or rolled back. This is the highest level of isolation. In addition to the isolation provided in 1 and 2, this level prohibits any transaction from writing data to records you have read until you commit or roll back the transaction.

Isolation levels 1, 2, and 3, all require more resources than non-transaction-oriented file update. At level 3 all the data used in a transaction is locked away from the use of any other transaction until the current transaction is completed. If the transaction is lengthy, it will tie up the data and resources and slow down overall system performance. It is important in high volume systems to make the transaction as tight as possible to minimize the performance impact.

Once a transaction handle has been obtained, the JOPEN verb is used to tie the journaled files to the transaction. The syntax for the JOPEN verb is

```
JOPEN(channel{,TID=int{,ERR=stmt})fileid
```

Where TID is the transaction ID obtained with the TOPEN function. If no TID is specified, the journaled file is opened in "auto commit" mode. In this mode an implied commit is executed after each update to the file. This is inherently slower, but it allows existing code to use journaled files without adding explicit TCOMMIT and/or TROLLBACK statements. Therefore, all updates to journaled files are transactions, either implicitly, as in auto-commit mode, or explicitly, when the TID is specified to bind the updates to a transaction. When journaled files are opened using a TID, they are now part of a new entity, the transaction.

Transactions must be committed or they will be automatically rolled back when a BEGIN, START, END, etc., are executed to close the journaled files. This guarantees that only the transactions that go through all the updates steps specified in a program, including the commit, are updated as a whole. It also guarantees that in the case of a system problem, incomplete transactions are automatically rolled back.

Transactions maintain the logical integrity of related data files. The journaled files used in a transaction cannot get out of synch, because the transaction operation groups the individual operations into a higher level, atomic operation. Consider a sales invoice update program. Each invoice generates updates to many files, such as Accounts Receivable, Sales Analysis, Inventory, Order Processing, and General Ledger. Without Transaction Tracking, the program could have updated only the A/R and S/A files. A system problem at this point would leave the other files out of synch with A/R and S/A. With Transaction Tracking, this cannot happen. The transaction would be rolled back, and the individual files would be in synch.

Transactions can be very complex. It is important, for performance reasons, to ensure that the definition of the "atoms" does not get too large. For example, you could set up a transaction that updates sales invoices, and does not commit anything until the entire set of invoices has been processed. This would be an enormous transaction and would in all likelihood tie up a lot of the system while every other process waited for a commit. Defining the transaction at the invoice level, where a commit is executed as each individual invoice is processed, including all the updates to the various files, is better. It preserves the integrity of the individual invoice updates, without unduly limiting other processes on the system.

We will close with an example of a simple invoice update. The update is limited to reading the invoice file and updating inventory and the A/R invoice file.

```
rem update invoices
rem read invoice file, and update inventory, and accounts receivable
rem treat updates as transactions

rem get transaction id
    InvoiceUpdateID=TOPEN(3)

rem open files for transaction, with maximum protection for concurrency
    InvoiceHdr=unt
    JOPEN(InvoiceHdr,TID=InvoiceUpdateID) "InvoiceHdr"

    InvoiceDet=unt
    JOPEN(InvoiceDet,TID=InvoiceUpdateID) "InvoiceDet"

    Inventory=unt
    JOPEN(Inventory,TID=InvoiceUpdateID) "Inventory"

    ARInvoice=unt
    JOPEN(ARInvoice,TID=InvoiceUpdateID) "ARInvoice"

rem data templates
dim InvoiceHdr$:"invoice_num:c(7),cust_num:c(7),invoice_date:n(9 ),
:order_num:c(7),invoice_total:n(8), discount_total:n(8)"
dim InvoiceDet$:"invoice_num:c(7),line_num:c(5),product_id:c(7),
```

```

:qty_ord:n(8),qty_ship:n(8),price:n(8)"
dim Inventory$:"product_id:c(7),description:c(30), gl_num_sales:c(7),
:gl_num_cost:c(7),uofm:c(4),price:n(8),cost:n(8),
:qty_on_hand:n(8),qty_committed:n(8),qty_sold_mtd:n(8),
:qty_sold_ytd:n(8)"

dim ARInvoice$:"cust_num:c(7),invoice_num:c(7),invoice_date:n(7),
:invoice_total:n(8),discount_total:n(8),amount_paid:n(8),date_paid:n(7)"

rem read invoice header
Next_Invoice:
    readrecord(InvoiceHdr,end=EOJ)InvoiceHdr$
rem read invoice detail and distribute
    readrecord(InvoiceDet,dm=*NEXT,key=InvoiceHdr.invoice_num$)
    readrecord(InvoiceDet,end=Next_Invoice)InvoiceDet$
    if InvoiceDet.invoice_num$<> InvoiceHdr.invoice_num$ then
:        goto Next_Invoice
rem update inventory
    extractrecord(Inventory,key=InvoiceDet.product_id$)Inventory$
    Inventory.qty_on_hand=Inventory.qty_on_hand- InvoiceDet.qty_ship
    Inventory.qty_sold_mtd=Inventory.qty_sold_mtd+
:        InvoiceDet.qty_ship
    writerecord(Inventory)Inventory$
rem A/R update - create a/r invoice
    ARInvoice.cust_num$=InvoiceHdr.cust_num$
    ARInvoice.invoice_num$=InvoiceHdr.invoice_num$
    ARInvoice.invoice_date=InvoiceHdr.invoice_date
    ARInvoice.invoice_total=InvoiceHdr.invoice_total
    ARInvoice.discount_total=InvoiceHdr.discount_total
    ARInvoice.amount_paid=0
    ARInvoice.date_paid=0


    writerecord(ARInvoice)ARInvoice$
rem commit the update
    tcommit(InvoiceUpdateID)

    goto Next_Invoice
rem end of job
EOJ:
rem close and clear invoice header and detail files
    close(InvoiceHdr)
    initfile "InvoiceHdr"

    close(InvoiceDet)
    initfile "InvoiceDet"

release

```

Note: Before creating your first journaled files, you must configure the Journaled File System from the Enterprise Manager. 

www,

www.basis.com/online/docs/documentation/index.htm