CHAPTER 14

Index-Organized Tables

Imagine you are an analyst in a large financial institution, advising clients on how to spend millions of dollars on buying and selling shares. To support this business you have a database that loads the trading volume and closing prices from all the major stock exchanges around the world at the close of every day. You also have a subtle and sophisticated front end that can take the prices of half a dozen shares during the last two years and can draw complicated charts that help you to decide when to buy and when to sell. (In fact, the analytic functions introduced in Oracle 8.1.6 [see Chapter 23] mean that the software need not be all that sophisticated.) Inevitably you have one of three options: an amazing piece of hardware, a cunning database design, or a performance problem.

Every day the pricing information for some 10,000 different shares is crammed into your database. This is likely to take approximately 1MB per day of storage in a table. But this means that if you pick one specific share, yesterday's price is going to be stored 1MB away from today's price, and the previous price is going to be stored 1MB away from yesterday's price, and so on down the line (Figure 14-1).

So if you execute a query to get two years of prices for a single share, you are going to visit approximately 450 different locations in the table that are so far apart that every single price is likely to require a real physical disk read. Because all but the newest disks rarely allow more than 30 I/Os per second (whatever the notional specification), your request for a single set of share prices is likely to take more than 15 seconds to return results. A chart comparing six sets of share prices could take more than 90 seconds to appear. A response time like that is unlikely to appeal to the typical end user.

Obviously there were tricks and design strategies to address this problem in earlier versions of Oracle—a single-table index cluster was the one most likely to help. In the absence of clustering, this is also a good example of why a

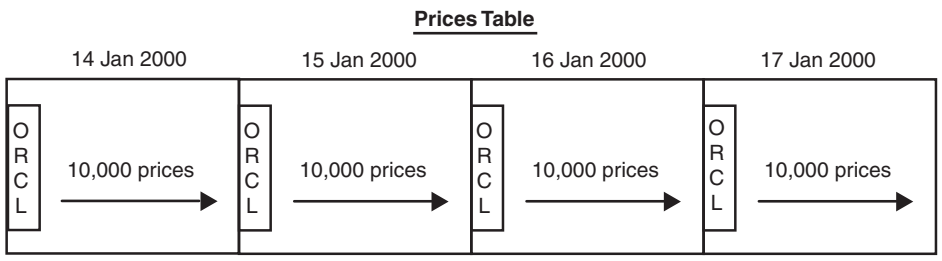

Figure 14-1. Extreme data scatter.

2KB block size can sometimes be a good idea to optimize the use of memory. However, in Oracle 8, the IOT is the official, efficient, and simplest solution.

Basic IOTs

We have already seen something of ordinary B-trees. Descending through the branches we finally reach the correct leaf, where we find a pointer to the physical location in a row in a table. An IOT in its simplest form does away with the last step completely. There is no separate table; the data for the row is stored in the leaf of the index itself. The code to create our share price table might look like this

```
create table share_history(
        ticker_code     varchar2(32)  references share_codes,
        trade_date      date,
        price_close     number(15,2),
        trade_volume    number(10),
        price_high      number(15,2),
        price_low       number(15,2),
        constraint sh_pk primary key (ticker_code, trade_date)
)
organization index
tablespace INDX
compress 1
;
```

The first point to note is that an IOT *must* have a primary key, and this primary key cannot be disabled, dropped, or deferred. Any attempt to do so

results in Oracle error "ORA-25188: cannot drop/disable/defer the primary key constraint for index-organized tables."

If you look in view family xxx_TABLES, you will find a row for the SHARE_HISTORY table, but the row holds some very strange values. For example, there is no tablespace, no value for INITIAL_EXTENT, and so on through all the other storage parameters. What you will find is that the IOT_TYPE column identifies the table as an IOT.

If you then look in the view family xxx_INDEXES, you will find an index called SH_PK, just as you would expect from the named primary key, but the index type is given as 'IOT=TOP'. If the primary key had not been named, you would have found an index with a name of the form SYS_IOT_TOP_23144, where the numeric section is the object number for the "nonexistent" table. Checking further through the view family xxx_SEGMENTS, you will find that there is no data segment to hold the SHARE_HISTORY table. There is only a segment of type index to hold our primary key index.

The second point to note from the SQL code defining the table is that the location of the index is dictated *not* by a USING INDEX clause, which you would normally associate with the primary key, but by the TABLESPACE clause declared for the table. In fact, you always have to bear in mind that although this object is visibly declared as a table, it is actually an index. Additional consequences of this are that the PCTUSED clause is illegal, and the PCTFREE clause is fairly pointless unless you are doing a rebuild or a CTAS. For the same reason, if you execute ***analyze table XXX validate structure***, you get a set of results in the INDEX_STATS view describing the primary key index.

The third point in the defining SQL code is the COMPRESS clause. This is a new feature of Oracle 8.1 that is designed to save space in indexes in general and in IOTs in particular. When the leading columns of an index are subject to frequent repetition you can tell Oracle to store the leading values once per leaf block rather than once per entry (see Chapter 7 for more details). In the case of the SHARE_HISTORY table, I collected the price for Oracle shares for the last 15 years (some 3,500 entries with a stock code of ORCL). Because the entire row is only approximately 40 bytes long, I should be able to get roughly 100 rows in a 4KB block, but by storing the stock code ORCL (and its overhead) only once per block, I save an additional 500 bytes per block, giving me room for an extra dozen rows in each block.

An added bonus of IOTs is that the typical query automatically accesses the data in the correct order, so Oracle does not need to sort the data after acquiring it:

```
select *
from    share_history
```

```
where   ticker_code = 'ORCL'
and     trade_date between to_date('01-Jan-1998')
                       and to_date('01-Jan-2000')
order by
            trade_date
;
```

My choice for an example of IOTs is very good from one perspective: It is a realistic case for which we need to turn the data around between input and output. However, because I chose to include only a few columns in the table, and then compressed one of them, I concealed one of the penalties of using IOTs. You don't normally get many rows to a block. In a normal index, a *row entry* consists of a primary key plus a rowid. In an IOT, a row entry really is the whole row. If the rows are rather long rows, you do not get very many of them in each leaf block, and this is a bit of a shame because it counters some of the benefits we hope to gain from packing lots of prices for a single stock into a single block. IOTs accommodate this by allowing you to split the nonkey columns into high-use columns and low-use columns. The low-use columns can then be relegated to an "overflow" area.

An IOT can be split into frequent use *and* occasional use *to get maximum benefit from the index-ordered packing.*

Assume that all our analysts are very interested in the closing prices but rarely look at the other prices and volume. We could relegate these columns to an overflow area by adding to our definition the clauses

```
including price_close
overflow tablespace OTHER_BITS
```

This tells Oracle two things. First, that we want to include all the columns up to and including the PRICE_CLOSE column in the index segment, and that the remainder should be stored in the overflow segment. Second, that the overflow segment should be stored in the OTHER_BITS tablespace. If we wanted to we could also add more physical storage details, such as extent sizes, to the OVERFLOW clause.

There is an alternative way to dictate which parts of a row should go to the overflow—the PCTTHRESHOLD clause. The PCTTHRESHOLD defaults to 50, and has to be between 1 and 50. If, as it is inserted or updated, a row

exceeds this fraction of the block size (allowing a little for overhead), then the columns causing the excess are pushed into the overflow segment. Conversely, if an update shrinks a row to less than the PCTTHRESHOLD, then the trailing columns migrate back into the "top" section of the IOT. For example, consider the following column definitions in a database with a 4KB block size:

```
create table art_catalog (
        object_id   number  primary key,
        purchased   date,
        description varchar2(500),
        history     varchar2(500)
)
organization index
pctthreshold 10
overflow tablespace other_bits
;
```

With a PCTTHRESHOLD of 10 and a block size of 4KB, Oracle decides that any row larger than approximately 400 bytes has to have its tail end moved to the overflow. If we insert a row with 200 bytes of description and 200 bytes of history, then just the history is pushed to the overflow. If we insert a row with 450 bytes of description, then the description and the history would be pushed to the overflow.

Personally, I would not use the PCTTHRESHOLD method because it implies that I do not really know what my data looks like or how it will be used. It also introduces a degree of uncertainty about where the I/O might take place, and this is always undesirable.

In fact, there is a little trap with PCTTHRESHOLD. As we saw in the very first code sample, it is possible to create an IOT without an overflow. However, if the nominal maximum size of the row definition (including various overhead) exceeds the "free space times PCTTHRESHOLD," then Oracle insists that you create an overflow segment. In the previous example, if you had defined the history and description columns to 1,500 bytes each, totaling 3,000 bytes out of a 4,000-byte free space, then Oracle would insist that you create the table with an overflow segment. If you ever include several large but optional columns in a table knowing that any one row will use only a few of them, you could find that Oracle insists that you create an overflow segment that never gets used. Unofficially, there is a way around this problem. If you set event 28657, which exists to protect Oracle when importing into a database with small blocks an IOT exported from a database with large blocks, you can break the PCTTHRESHOLD rule without raising an error.

If you want to check the storage details of this overflow segment, by the way, you will find it in the USER_TABLES view, with a name like SYS_IOT_OVER_23144. Again, the number corresponds to the object number of the nonexistent table. Unfortunately, it is not yet possible to specify a meaningful name for an overflow segment. If you try to use ***alter table rename*** you get Oracle error "ORA-25191: cannot reference overflow table of an index-organized table."

Because the overflow segment effectively takes the role of a traditional table, it does make sense to think about using the PCTUSED and PCTFREE parameters on the overflow table. This can result in a slightly odd looking piece of SQL code that appears to have two separate storage clauses:

```
create table share_history(
    ticker_code     varchar2(32),
    trade_date      date,
    price_close     number(15,2),
    price_high      number(15,2),
    price_low       number(15,2),
    trade_volume    number(10),
    constraint sh_pk primary key (ticker_code, trade_date)
)
organization index
tablespace indx
storage (initial 500k next 500k pctincrease 0)
compress 1
including price_close
overflow tablespace other_bits
storage (initial 1m next 1m pctincrease 0)
pctfree 2
pctused 98
;
```

One little trap to watch out for is that the LOGGING attribute of an IOT and its overflow must be the same (unlike an ordinary table and its indexes). If the INDX and OTHER_BITS tablespaces in the previous SQL code happened to have different default LOGGING attributes, then Oracle would raise the error "ORA-28662: IOT index and overflow segments must share the same LOGGING attribute." Having said this, you could use the command

```
alter table share_history move
overflow tablespace somewhere_else
nologging
;
```

Although there is no legal path in the SQL reference manual for this command, it does work. If you use it you could end up with the two segments in different LOGGING states. This does highlight a particularly annoying issue with the current state of the manuals: There are some combinations of command options that appear to be legal and fail; there are others that appear to be illegal but work to one degree or another. The previous example is one that seems reasonable, could be used "by accident," and will get your database into a state that is notionally unsupported. Be very careful if you discover something that appears to work, but is not documented. Check the legality with Oracle Support before you use it in a production system.

Indexes on IOTs

One of the limitations on IOTs in Oracle 8.0 was that you could not create any other indexes on them. B-tree indexes store the rowid of each row, and bitmap indexes use rowid ranges to produce maps. But a rowid is the physical location in the database of a row (expressed as the file, block in file, and row position in block), and the rows in an IOT can move from block to block as new rows are created and blocks get split. IOTs do not have permanent, physical rowids.

In Oracle 8.1, secondary indexes are allowed through the creation of a new data type—the UROWID or universal (logical) rowid. UROWIDs can hold rowids for all types of objects, including non-Oracle data objects. In particular, they may hold data that represents locator information for a row in an IOT.

Secondary indexes can be unique, nonunique, or even function based, but they may not be bitmap indexes because bitmap indexes require an absolutely unmovable physical location for the rows in the map. An odd little detail of secondary indexes is that INDEX_TYPE in the view family xxx_INDEXES is marked as FUNCTION-BASED NORMAL.

For IOTs, one of the items of information that the UROWID holds is the value of the primary key. The strategy for a query on the secondary index is that Oracle should traverse the secondary index to find the related primary keys, and then traverse the IOT to get to the data (Figure 14-2). In effect this means that a query against a secondary index on an IOT does roughly twice as much logical I/O as the identical query against a normal table.

In an attempt to reduce this workload, however, the UROWID also contains a "guess" of the block where the row should be. In fact, this guess is simply the address of the block where the row was located when the index was built or where the row entry was first inserted.

Of course, because rows can move from block to block in IOTs, this guess is likely to go out of date with the passing of time. Because it is *never*

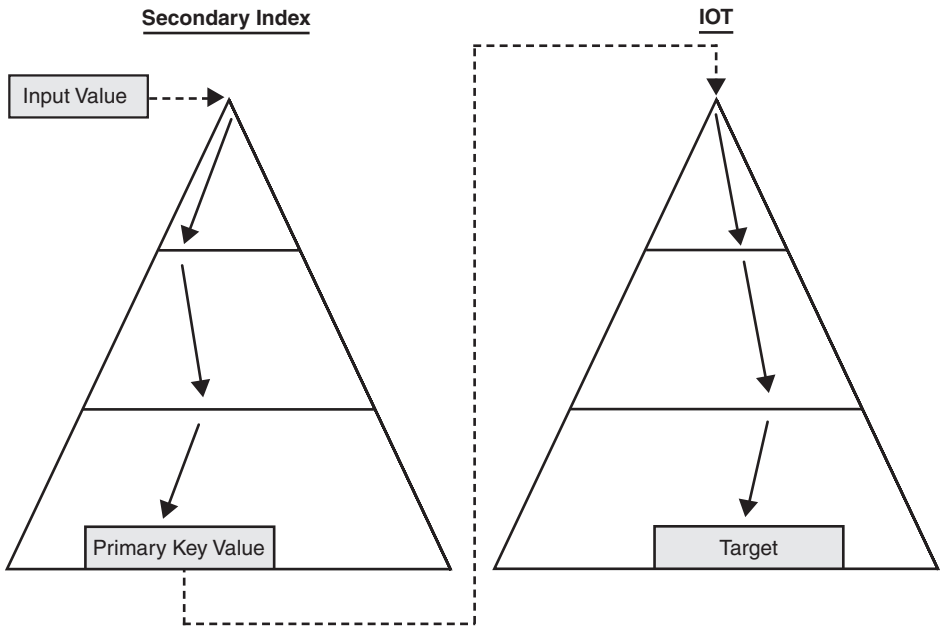

Figure 14-2. Secondary index use on IOTs.

corrected on the fly, you could end up with a very unpleasant performance overhead as Oracle uses the guess to jump to the wrong block and then goes back to traverse the IOT to find the right block.

The optimizer has some degree of understanding about guesses, and there is a new statistic called PCT_DIRECT_ACCESS in the view family xxx_INDEXES, which is the estimate of the percentage of the guesses that are likely to be correct. This information is probably included in the optimizer's decision to use or ignore the guesses as well as the basic decision of whether to use the secondary index at all. This value is set to 100 when you create the secondary index, and is set back to zero when you rebuild the IOT.

You may think that rebuilding the secondary index (ALTER INDEX REBUILD) would set PCT_DIRECT_ACCESS back to 100. In fact, the *Oracle 8i Concepts* manual does state quite explicitly that "rebuilding a secondary index on an index-organized table involves reading the base table, unlike building an index on an ordinary table." However, this doesn't seem to be entirely true. I have come across cases when Oracle appeared to rebuild the index without reference to the table, and without resetting

PCT_DIRECT_ACCESS. It may be necessary to drop and re-create your secondary indexes to get the guess quality back up to 100%.

Of course, because the secondary index contains the indexed column, the primary key, and a block guess (and a few bytes of extra overhead), it is likely to be significantly larger than the equivalent index on an ordinary table. On the other hand, when you rebuild an IOT the secondary index is *still usable* and does not have to be rebuilt. Because the IOT can be rebuilt on-line, while its secondary indexes stay valid, this offers some interesting possibilities to sites that have to run 24/7. As always, it's a trade-off: some wasted space and a built-in reduction in performance against nonstop operation.

Another side effect of the way in which secondary indexes contain the primary key is that you have a bonus execution path in queries of the form

```
select primary_key_columns
from index_organized_table
where {secondary_indextest is true}
```

At first sight this looks like a simple index access to a table requiring two steps. However, the secondary index contains the primary key columns, so the execution can be just a single-step range scan on the secondary index.

There are a few anomalies about secondary indexes and guesses. First, I'm not sure that Oracle actually uses the guesses, notwithstanding the comments in the manuals. Perhaps it's just a question of data size and statistics, but in all the tests I have done to date, Oracle has always scanned the secondary index then traversed the IOT without jumping to a guess block. (You have to shut down the database and probably reboot the system to be completely sure of this.) Second, it is arguable that the concept of using a guess is a bad idea anyway, as seen in Figure 14-3.

Remember that B-trees actually fan out quite quickly. For clarity and convenience, the diagrams you find in books tend to show each block with perhaps three or four children, but in real Oracle databases it is common for each block to have a hundred or more children. Consequently, it is eminently possible for the branch blocks of an index to be permanently buffered in memory, whereas the leaf layer is so large that it is hardly buffered at all. (I actually read a very interesting postgraduate dissertation recently that came to a fascinating conclusion, but based its entire mathematical argument on the fact that each branch in a B-tree would have roughly four leaf blocks.)

Look at Figure 14-3 and consider what will happen if (1) the leaf layer is not buffered, and (2) we use the guess. We hit two buffered blocks and do one physical read in the secondary index, then read the guess and jump to the

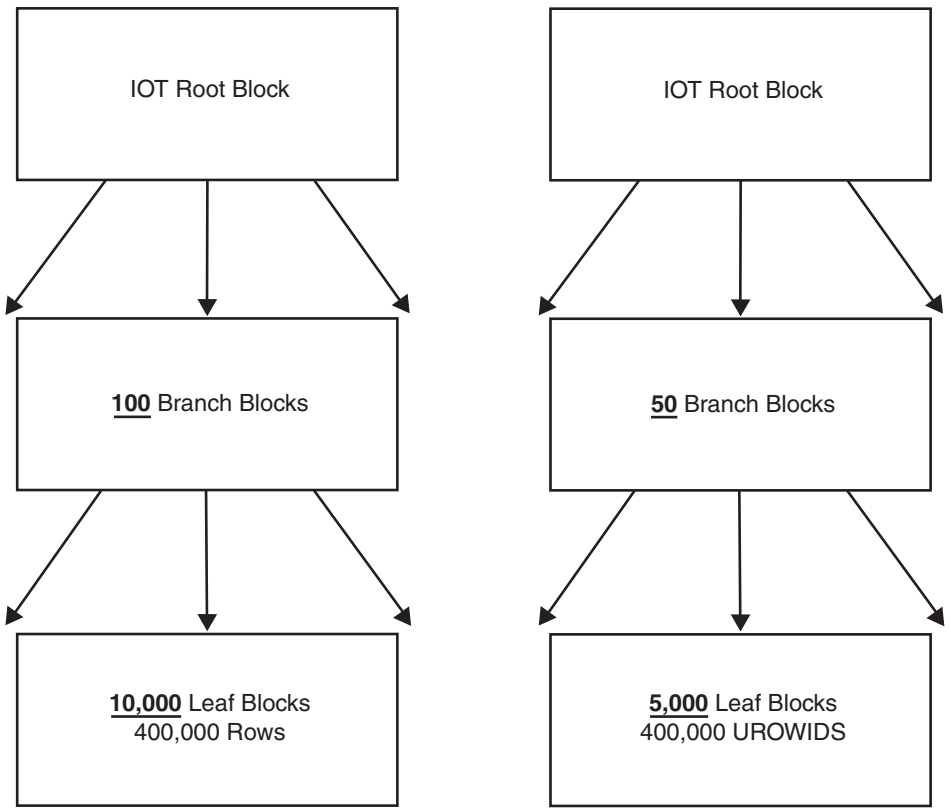

Figure 14-3. Fanning out of B-trees.

primary. If the guess is right, we do one more physical read. If the guess is wrong, we do two more physical reads.

Now assume we ignore the guess. We hit two buffered blocks and do one physical read in the secondary index. We use the primary key to do two buffered reads in the IOT, and then do one physical read to get the IOT leaf.

In summary, if the guess is good we save two logical I/Os (perhaps one ten thousandth of a CPU second). If the guess is bad we add one physical read (at least one fiftieth of an elapsed second) and all its overhead. Is this a trade even worth considering? In general I suspect not. It seems to me that you really need to guarantee an almost perfect success rate on guesses before the trade is worthwhile.

On-line Moves

One of the important things to remember about IOTs is that they are indexes. So when you do anything to the table, it is worth considering if there are any special side effects that may appear because you are really handling an index.

One of the particular options to consider is rebuilding tables. An ordinary table cannot be rebuilt on-line, but an ordinary index can. So the following command will work, and the table will be built on-line while it is still fully usable:

```
alter table iot_tab move online;
```

Of course it has the same restriction as on-line rebuilds of ordinary indexes. The rebuild cannot start until there are no active transactions on the table, and the rebuild cannot complete until there are no active transactions on the table. Furthermore, if you have an overflow segment, it is not rebuilt at the same time.

However, it is always worth looking a little closer at IOTs. If you execute this command with SQL_TRACE switched on, you will find the following occurring under the covers:

```
create unique index "JPL1"."IOT_TAB_PK" on "IOT_TAB" (OBJECT_ID)
index only toplevel
move
as
select * from "IOT_TAB"
```

And if you examine your session statistics, you may find that to build this index, Oracle does a fast full scan of the existing table and sorts the entire dataset. In all the tests I performed, Oracle did not use a range scan to read the data in sorted order. Be careful about moving a large IOT on-line on a production system; the overhead is high.

There are other catches to using IOTs that revolve around the fact that they are really indexes. You can do a direct SQL*Load into an IOT, but because the IOT is an index, Oracle loads the new data, sorts it, then copies the whole of the old IOT to merge it with the new dataset—a fact that is not obvious if you think of the IOT as a table. In a similar vein, a NOLOGGING table allows you to INSERT /*+_ APPEND */ without generating any redo log. But if the table is actually an IOT, then the logging takes place. Inserts to an index are always logged, even an index created in a NOLOGGING tablespace.

Advanced IOTs

IOTs can be partitioned, can include LOBs and varrays, and can contain nested tables—three restrictions that have been lifted since version 8.0. There is still a little way to go in combining these features, because *partitioned* IOTs cannot include LOBs, varrays, and nested tables.

Because the obvious purpose of IOTs is to rearrange data at load time for subsequent rapid access while avoiding storage overhead, I would think very carefully before including any variant of these large-object types inside an IOT. I don't see the restrictions on partitioned IOTs as a great defect, but I would expect the basic partitioning option to be very useful.

LOBs, Varrays, and Nested Tables

Most of the issues of dealing with "large" objects are addressed in other chapters. However there are a couple of issues specific to IOTs.

If you really want to include LOBs or large varrays in pure IOTs, Oracle rather sensibly stores them out-of-line, irrespective of how small the actual LOB/varray content may be. If you include an overflow segment, then you can ***enable storage in row***, as long as the LOB/varray is defined to be part of the overflow.

If you include a nested table in an IOT, then the issues are no different from including a nested table in a normal table. A more interesting situation arises when you consider the fact that any nested table should probably be an IOT anyway. After all, the purpose of a nested table is to associate an open-ended collection of objects with its parent, so that parent and collection can be accessed rapidly. The nested table as an IOT seems ideal. There are a couple of problems though. One problem is that you cannot specify an overflow segment in the NESTED TABLE clause. For example,

```
create type jpl_row as object (
        n1 number(6),
        v1 varchar2(1000),
        v2 varchar2(1000)
);
/

create type jpl_table as table of jpl_row;
/
```

```
create table nest_demo(
        Id          varchar2(32) primary key,
        nest1       jpl_table
)
nested table nest1 store as nested1 (
        (primary key (nested_table_id, n1))
        organization index
        compress 1
);
```

There are two points to notice especially with this ***create table*** statement. First, I have to declare a primary key with an IOT, so I have used the implicit NESTED_TABLE_ID column of the nested table combined with one of the real nested table columns. Second, because all the nested items for a single parent will be very close to each other, and the 16-byte NESTED_TABLE_ID will be the same for any one parent, I have eliminated repetition of that ID by using the COMPRESS option on the first column of the primary key.

The most important point, though, is one that is not immediately obvious. Because I have a 4KB block size, a full-size row of the jpl_row type (2,000+ bytes) would have exceeded 50% of the block. Because PCTTHRESHOLD has a default value of 50%, I should have got Oracle error "ORA-01429: index-organized table: no data segment to store overflow row-pieces" when I tried to create my nested table as an IOT. This did not happen. Instead, the error message only appears when I try to put in some data that needs to overflow.

I need an overflow segment for the nested IOT, but I can't create it when I create the rest of the table. As with so many of the new, more complex features of tables, you have to create the table and then issue an ***alter table*** command immediately afterward. In this case, to avoid the rather sneaky and unfair runtime error message, you have to add the overflow with the ***alter table*** command, altering the *nested table*:

```
alter table nested1
add overflow
tablespace users;
```

Unfortunately, when I actually tried to do this, I got an error message telling me that I had no privilege on the SYSTEM tablespace. When I gave myself the privilege, the overflow segment duly appeared in the SYSTEM tablespace despite the fact that it was not my default tablespace. Nominally it is possible to move an IOT, including the overflow, with the ***alter table***

command, but when I tried it I got Oracle error ORA-03113. These are known bugs and are likely to be fixed by version 8.1.6.

It has to be said that when you start taking complex and convoluted routes through the descriptions of the ***create table*** and ***alter table*** commands in the SQL reference manual, it is possible to come up with combinations of features that the Oracle code doesn't handle very well.

The other problem with nested tables and IOTs is an internal one that affects performance. When you create a nested table (see Chapter 17), Oracle is supposed to create a unique index on a hidden column of the parent table. Of course, for an IOT this index would be a secondary index rather than an ordinary index. But it does not get created, so there is no efficient access path from the nested table to the parent table, and all queries have to drive through the parent table.

Partitioning

IOTs can be partitioned, but only by range (not hash), and the partitioning columns must be a subset (although not necessarily the leading columns) of the primary key index. There are also some restrictions on the structure of the IOT if it is partitioned. For example, LOBs and varrays are not allowed. Perhaps the most significant restriction is that a partitioned IOT cannot be rebuilt on-line, even one partition at a time. On the plus side, though, if you rebuild a partition of a partitioned IOT, even the global indexes and globally partitioned indexes remain usable (see Chapter 12).

As an example of why we might want to partition an IOT, let us go back to the share pricing system. We decided to create an IOT based on the ticker code and the trade date because most users wanted to query a small number of ticker codes over a range of dates. However some users may, from time to time, want to take a snapshot of a significant fraction of the market on a given date, perhaps to derive some type of in-house market index. Without some alternative access path, this will be a *very* expensive query.

One option is to create a secondary index on the trade date, but this would probably still result in a very large volume of physical I/O because the prices for a given date would be very widely scattered across a very large table. Therefore, we could try partitioning the data by trade date as follows:

```
create table share_history(
        ticker_code     varchar2(32),
        trade_date      date,
        price_close     number(15,2),
        price_high      number(15,2),
```

```
        price_low       number(15,2),
        trade_volume    number(10),
        constraint sh_pk primary key (ticker_code, trade_date)
)
organization index
tablespace indx
compress 1
including price_close
overflow tablespace spare_bits
partition by range (trade_date)(
partition sh_1984_jan values less than
        (to_date('01-feb-1984','dd-mon-yyyy'))
        tablespace users overflow tablespace users,
. . .
partition sh_1999_oct values less than (maxvalue)
)
;
```

This produces a table with approximately 400 segments (16 years at 12 partitions per year) and 2 segments (1 IOT top and 1 IOT overflow per year). Every single segment could be in a separate tablespace if required. The code shows how to put the January 1984 segments into specific tablespaces.

If you took an approach like this, a query for all the prices for 14 January 1999 would have the option to scan just one month's data, which may be much more effective than collecting every ten thousandth row from the table by using a secondary index.

Restrictions on IOT Partitions

As you might expect, it is possible to use the EXCHANGE PARTITION feature with partitioned IOTs. You just have to be a little fussier when doing it. The table with which you are exchanging must be an IOT. If either object has an overflow segment, then both must have an overflow segment. The primary key must have the same columns in the same order. The level of compression of the keys must be the same. Unfortunately, this is not checked as the exchange takes place, so if there is a mismatch, subsequent queries can crash with ORA-00600 errors, so be careful. If you discover the error in time you can exchange back without any harm.

The impact of index-based constraints on exchanging partitions (see Chapter 12) applies to IOTs because an IOT is built on its primary key. If you expect to use the EXCHANGE PARTITION feature on a partitioned IOT, don't forget to set the primary key to ENABLED NOVALIDATE. Unlike

normal partitioned tables, it is not possible to merge IOT partitions. If you think this is important, you will have to create a procedure to exchange the partitions and drop one, then build a new IOT from the two exchanged partitions before swapping it back in.

Problems and Quirks

There are still quite a few bugs with IOTs in version 8.1.5 (as witness my previous nested table example). A simpler example shows up when trying to use the COMPUTE STATISTICS options of rebuilding an index: For an IOT or its secondary indexes, this results in the process crashing.

Another, perhaps more significant bug, is that referential integrity does not work properly if the parent table is an IOT. It is possible to delete a parent row and leave a collection of orphans, and ON DELETE CASCADE does not work. This may not affect many systems because IOTs are more likely to be child tables than parent tables. The bug has been fixed in version 8.1.6.

One of the bugs that is not fixed in version 8.1.6 shows up as a potentially serious defect in partitioned IOTs. It is very easy to get library cache deadlocks occurring during partition maintenance. For example, if two sessions try to split two different partitions of the same IOT at the same time, the one that starts second waits for the first split to complete, and then fails with error "ORA-04020: deadlock detected while trying to lock object XXXX." (Splitting IOT partitions is also the special case when partitions of secondary indexes become unusable.)

This would probably be bearable, and forgivable, if it occurred only for concurrent DDL, but unfortunately the problem is much worse: Because of an error in the acquisition of KGL (library cache) locks during IOT partition maintenance, even a simple ***select*** statement that starts while partition maintenance is in progress waits and then fails with a 4020 error.

Added to the problem of functional bugs is the list of errors in the manuals. If you discover that you can't get some feature suggested by the manual to work, it is likely that the manual is wrong. For example, the section in the SQL reference manual on ALTER INDEX suggests that you can rebuild a secondary index of an IOT on-line. But when you try, you get error "ORA-08108: may not build or rebuild this type of index on line."

When you rebuild the table on-line, only the IOT top segment is rebuilt; the overflow and LOB segments cannot be rebuilt on-line. This is not surprising because the IOT top segment holds a pointer to the absolute physical location of data in the other two segments.

Oracle insists that you have an overflow segment if the sum of the declared column sizes (plus some overhead) totals more than 50% of the free space for the block. You cannot override this, officially, even if you know that your data will never use the full space allowed by the column definitions.

Although the table can be rebuilt on-line with the ***alter table move*** command, it has the same problem as a normal index on-line rebuild (***alter index rebuild***). There has to be a couple moments when there are no uncommitted transactions on the table—one so that the rebuild can start and one so that the rebuild can complete. On a high-use, international, 24/7 Web site, this may make it impossible to do an on-line rebuild.

The secondary indexes on an IOT cannot be rebuilt on-line, despite comments to the contrary in the manual. In some cases this may eliminate the benefit of being able to rebuild the IOT itself on-line.

An IOT is an index, so even when you define it to be NOLOGGING, it is not possible for Oracle to add data to it without changing existing blocks. Thus, rollback and redo are always generated for IOTs. The direct-load hint /*+ APPEND */ doesn't do anything. The only exception to this behavior occurs when a complete build (CTAS or ***alter table move***) takes place.

There are some restrictions on using IOTs for materialized views (see Chapter 23). An obvious one is that the materialized view log cannot USE ROWID because IOTs do not have rowids (not that you should be using rowids with materialized views anyway). A less obvious restriction is that certain types of materialized views (for example, aggregate views) simply may not be created as IOTs. Because materialized views provide such enormous performance gains, this restriction is unlikely to be of great significance.

Strategy Notes

IOTs allow data to be packed in a different order from the order in which it naturally arrives. This slows down data entry, but it can enhance data retrieval immensely.

IOTs can be rebuilt on-line using the ***alter table move*** command, provided they are not partitioned and that you do not want to rebuild the overflow segment. Oracle seems to sort the entire table to do this though.

If index organization is relevant, then the table is likely to be of a very simple structure with very short rows. I doubt if there is any significant performance benefit to be found in applying IOT technology to tables that contain frequently used LOBS or varrays.

Nested tables are likely to be very good candidates for IOTs. They are likely to have small rows, and the data for a given parent row is likely to be accessed as a single unit at all times.

If IOTs are likely to be beneficial, then they tend to be suitable for index compression. This is particularly relevant for nested IOTs because there is a 16-byte ID column prepended to the nested rows.

It is sometimes possible to separate a table into FREQUENTLY ACCESSED and RARELY ACCESSED columns. If the table is an IOT you can choose to move the RARELY ACCESSED column to an overflow segment, thus reducing the size of the index and improving the efficiency of accessing the frequently used data.

IOTs can have secondary indexes. These use the primary key as a logical rowid but include a "guess" regarding the physical location of that row in the IOT. Because rows move from block to block in the IOT as the table grows, these guesses become stale and the secondary indexes become less efficient.

Although simple IOTs can be rebuilt on-line, any overflow or LOB segments cannot be rebuilt on-line, which rather reduces the benefit. There is also a restriction in version 8.1.5 that stops the secondary indexes from being rebuilt on-line.

If you have tables that have to be available 24/7, you may consider turning them into IOTs even though they are *not* natural candidates for this treatment, simply so that you can do an on-line rebuild from time to time. Of course, you may have to push all the nonkey columns into the main segment, and the table cannot be partitioned; otherwise the rebuild cannot be on-line. These restrictions should be weighed carefully against the benefit of being able to do the on-line rebuild.

IOTs are a very interesting addition to the tools available when building DSS-type systems in particular, but there are a number of limitations (often apparently minor ones) in how they can be used. I have mentioned only a few of the more significant ones in this chapter, so do make sure that you test every single assumption you make about IOTs before you commit large amounts of data to them.

If you think that IOTs may be of use, test everything with SQL_TRACE switched on. There are lots of strange bits of recursive SQL that appear when you start to dig into the features of IOTs.