Partitioning: What, When, Why & How

By Arup Nanda

Introduction

Partitioning is nothing new in Oracle Databases. There has been scores of books, articles, presentations, training sessions and even pages in Oracle manuals on the partitioning feature. While being serious sources of information, most of the texts seem to highlight the usage aspect of the feature, such as what type of partitioning, how to create a partitioned table or index and so on. The success of partitioning lies in the design phase. Unless you understand why to use a certain type of partitioning, you may not be able to articulate an effective strategy. Unfortunately this falls in the gray area between modeling and DBA, an area probably seldom visited and often neglected.

In this article, you will learn how to use partitioning to address common business problems, understand what is needed to in the design process, how to choose a specific type of partitioning along with what parameters affect your design and so on. It is assumed that you already know the concepts of partitioning and can get the syntax from manuals. After reading this, you will be able to address these questions:

- When to use partitioning features
- Why partition something, to overcome what challenges
- What type of partitioning scheme to choose
- How to choose a partition key
- Caveats and traps to watch out for

Learning is somewhat easier when illustrated with a real life scenario. At the end of the article, you will learn how these design decisions are made with a complete case study.

When

The partitioning skills require a mixture of Modeling and DBA skills. Usually you decide on partitioning right after logical design (in the domain of the Modelers) and just before physical design (in the domain of the DBAs). However, this is an iterative process. Be prepared to go back and change the logical design if needed to accommodate a better partitioning strategy. You will see how this is used in the case study.

A question I get all the time is what types of tables are to be considered for partitioning, or some variant of that theme. The answer is in almost all the cases for large tables. For small tables, the answer depends. If you plan to take advantage of partition-wise joins, then small tables will benefit too.

Why Partition?

The very basic question is ridiculously simple – why even bother partitioning a table? Traditionally these two have been the convincing reasons:

Easier Administration

Smaller chunks are more manageable than a whole table. For instance, you can rebuild indexes partition-by-partition, or move tables to a different tablespaces one partition at a time. Some rare usage includes data updates. When you update the entire table, you do not need counters to keep track of how many rows were updated to commit frequently. You just update one partition at a time.

Performance

This competes with the ease of administration as a top reason. When you perform full table scans, you are actually performing full partition scans. When

you join two tables, Oracle can automatically detect the data values being on one partition and choose to join the rows in different partitions of several tables – a fature called partition-wise join. This enhances the performance queries significantly.

Other lesser know performance enhancing features come from reduced latching. Partitioning makes several segments out of a single table. When the table is accessed, the segments could potentially be on multiple cache buffer chains, making fewer demands on the latch allocation.

Hot Indexes

Consider an index on some sort of sequential number – a monotonically increasing number. Since the numbers are added incrementally, a handful of leaf blocks may experience contention, making the index hot. Over period of time, the hot portion moves to a different part of the index. To prevent this from happening, one option is to create a hash partitioned index. Note, the table may or may not be partitioned; but the index could be – that's the beauty of hash partitioned index, introduced in Oracle 10g R2.

Here is an example of how it is created on a table called RES.

```
create index IN_RES_01
on RES (RES_ID)
global
partition by hash (RES_ID)
partitions 8
```

In this example the table RES is un-partitioned; while index is partitioned. Also, note the use of the clause "global". But this table is not partitioned; so global shouldn't apply to the index. Actually, it does. The global clause can also be used on partitioned indexes which are on unpartitioned tables.

This creates multiple segments for the same index, forcing index blocks to be spread on many branches and therefore reducing the concentration of access on a single area of the index, reducing cache buffer chain related waits.

Since the index is now partitioned, it can be rebuilt partition-by-partition:

```
alter index IN_RES_01 rebuild partition
<PartName>;
```

It can be moved to a different tablespace, renamed and so on, as you can with a regularly partitioned index.

More Important Causes

The previous two causes, while important, are not the only ones to be considered in designing partitioning. You have to consider more important causes.

Data Purging

Purging data is a common activity in pretty much any database. Traditional methods of purge rely on deleting rows, using the DELETE command. Of course, TRUNCATE command can be used to delete the whole table; but purge is hardly ever for the entire table. DELETEs are very expensive operations; they generate a large amount of REDO and UNDO data. To prevent running out of undo space, you may resort to frequent commits, which stress the I/O subsystem since it forces a log buffer flush.

On the other hand, partition drops are practically free. All you have to do is to issue a `ALTER TABLE TableName DROP PARTITION P1` and the partition is gone – with minimal undo and redo. The local indexes need not be rebuilt after the drop; but global indexes will need to be. From Oracle 9i onwards, you can use `UPDATE GLOBAL INDEXES` clause to automatically update the global indexes during partition drop.

Archival

A part of the purge process may be archival. Before dropping the data, you may want to store the data somewhere else. For instance, you are deleting some sales data for April 2008; but you want to move them to a different table for future analysis.

The usual approach is issuing `insert into archival table select * from main table` statement. However, INSERT is expensive – it generates a lot of undo and redo. You can reduce it somewhat by using the /*+ APPEND */ but you can't avoid the massive selection from the table.

This is where the power of partition exchange comes in. All you do is to convert the partition to a standalone table. In line with the example shown above, you will need to create an empty table called TEMP – the same structure as the SALES table; but not partitioned. Create all the indexes as well. After the creation, issue the following:

```
ALTER TABLE SALES EXCHANGE PARTITION APR08
WITH TABLE TEMP INCLUDING INDEXES
```

This makes the data in the former partition available in TEMP and the partition empty. At this time, you can drop the partition APR08. The table TEMP can be exchanged with the partition APR08 of an archival table; or just renamed.

During partition exchange, local indexes need not be rebuilt. Global indexes will need to be rebuilt; but can be automatically maintained if the UPDATE GLOBAL INDEXES clause is given. This is the fastest, least expensive and the preferred approach for archival.

Materialized Views Refreshes

You should already be familiar with Materialized Views, which are results of queries stored as segments, just like tables. The MV stores the data; not maintain it. So, it needs to be refreshed from time to time to make the data current. Traditionally, the approach to refresh the MV is calling the procedure REFRESH in the DBMS_MVIEW package.

There is nothing wrong with the approach; but it locks the entire MV until the refresh is complete. Also, the data is inserted using INSERT /*+ APPEND */ statement, which stresses the I/O subsystem.

Another approach is possible if the MV is partitioned properly. If done right, only a few partitions of the MV will need to be refreshed, not all. For instance, suppose you have an MV for Sales data partitioned monthly. Most likely the partition for a previous period is not going to change if you refresh it, as the base table data won't have changed. Most likely only the last month's partition needs to be refreshed. However, instead of refreshing, you can use the Partition Exchange trick.

Fist you create a temp table structurally identical to the MV but not partitioned, along with indexes, etc. You populate this temp table with the data from base tables. Once done, you can issue `alter table MV1 exchange partition SEP08 with table temp update all indexes` which updates the data dictionary to show the new data. The most time consuming process is building the temp table, but during the whole time the MV remains available.

Backup Efficiency

When a tablespace is made read-only, it does not change and therefore needs only one backup. RMAN can skip it in backup if instructed so. It is particularly useful in DW databases which are quite large and data is mostly read only. Skipping tablespaces in backup reduces CPU cycles and disk space.

A tablespace can be read only when all partitions in them can be considered unchangeable. Partitioning allows you to declare something read only. When that requirement is satisfied, you can make the tablespace read only by issuing `alter tablespace Y08M09 read only;`

Data Transfer

When you move data from one database to the other, what are the normal approaches? The traditional approach is issuing the statement `insert into target select * from source@dblink` or something similar. This approach, while works is fraught with problems. First, it generates redo and undo (which can be reduced by the APPEND hint). Next, a lot of data is transferred across the network. If you are moving the data from the entire

tablespace, you can use the Transportable Tablespace approach. First, make the tablespace read only. Then copy the datafile to the new server. Finally "Plug in" the file as a new tablespace into the target database. You can do this even when the platforms of the databases are different, as well. For a complete discussion and approach, refer to my Oracle Magazine article http://www.oracle.com/technology/oramag/oracle/04-sep/o54data.html.

Information Lifecycle Management

When data is accessed less frequently, that can be moved to a slower and cheaper storage, e.g. on EMC platforms from DMX to Clariion or SATA. You can do this in two different ways:

(A) **Partition Move**

First, create a tablespace called, say, ARC_TS on cheaper disks. Once created, move the partition to that tablespace using ALTER TABLE TableName MOVE PARTITION Y07M08 TABLESPACE ARC_TS. During this process, the users can select from the partition; but not update it.

(B) **ASM Approach**

While the tablespace approach is the easiest, it may not work in some cases where you can't afford to have a downtime for updates. If your datafiles are on ASM, you may employ another approach:

```
ALTER DISKGROUP
DROP DISK CostlyDisk
ADD DISK CheapDisk;
```

This operation is completely online; the updates can continue when this is going on. The performance is somewhat impacted due to the rebalance operation; but that may be tolerable if the asm_power_limit is set to a very low value such as 1.

How to Decide

Now that you learned what normal operations are possible and enhanced through partitioning, you should choose the feature that is important to you. This is the most important part of the process – understand the objectives clearly. Since there are multiple objectives, list them in the order of importance. Here is an example:

- Data Purging
- Data Archival
- Performance
- Improving Backups
- Data Movement
- Materialized View Refreshes
- Ease of Administration
- Information Lifecycle Management

Now that you assigned priorities, you choose the partitioning approach that allows you to accomplish the maximum number of objectives. In the process of design, you might find that some objectives run counter to the others. In that case, choose the design that satisfies the higher priority objective, or more number of objectives.

Case Study

To help understand this design process, let's see how decisions are made in real life scenarios. Our story unfolds in a fictitious large hotel company. Please note, the company is entirely fictional; any resemblance to real or perceived entities is purely coincidental.

Background

Guests make reservations for hotel rooms, for one or more number of nights. These reservations are always made for future dates, obviously. When guests check out of the hotel, another table CHECKOUTS is populated with details. When guests buy something or spend money such as order room service or buy a movie, records are created in a table called TRANSACTIONS. There is a concept of a folio. A folio is like a file folder for a guest and all the information on the guests stay goes in there. When a guest checks in, a record gets created in the FOLIOS table. This record gets updated when the guest checks out.

Partition Type

To understand the design process, let's eavesdrop on the conversation between the DBA and the Data Modeler. Here is a summarized transcript of questions asked by the DBA and answered by the Modeler.

Q: How will the tables be purged?
A: Reservations are deleted 3 months after they are past. They are not deleted when cancelled. Checkouts are deleted after 18 months.

Based on the above answer, the DBA takes the preliminary decision. Since the deletion strategy is based on time, Range Partitioning is the choice with one partition per month.

Partition Column

Since deletion is based on RES_DT and CK_DT, those columns were chosen as partitioning key for the respective tables.

```
create table reservations (…)
partition by range (res_dt) (
   partition Y08M02 values less than
(to_date('2008-03-01','yyyy-mm-dd')),
   partition PMAX values less than
(MAXVALUE)
)
```

Here we have chosen a default partition PMAX to hold rows that go beyond the boundary of the maximum value.

Access Patterns

Next, we want to know more about how the partitions are going to be accessed. The DBA's question and Modeler's answer continues.

Q: Will checkout records within last 18 months be uniformly accessed?
A: No. Data within the most recent 3 months is heavily accessed; 4-9 months is lightly accessed; 9+ months is rarely accessed.

Based on the above response, we decide to use Information Lifecycle Management to save storage cost. Essentially, we plan to somehow place the most recent 3 months data on highest speed disks and so on.

Access Types

To achieve the objectives of the backup efficiency, we need to know if we can make the tablespace read only.

Q: Is it possible that data in past months can change in CHECKOUTS?
A: Yes, to make adjustments.
Q: How likely that it will change?
A: Infrequent; but it does happen usually within 3 months. 3+ months: very rare.
Q: How about Reservations?
A: They can change any time for the future; but they don't change for past records.

This is a little tricky for us. Essentially, none of the records of CHECKOUTS can be made read-only, we can't make the tablespace read only as well. This affects the Information Lifecycle Management decision as well. So, we put on our negotiator hat. We ask the question: what if we make it read only and if needed we will make it read write? But the application must be tolerant to the error as a result of being read-only.

After a few rounds of discussions, we decided on a common ground – we will keep last three months of data read write; but make everything else read only. If needed, we can make it read write, but with a DBA's intervention. This decision not only improves the backup, but helps the ILM objective as well.

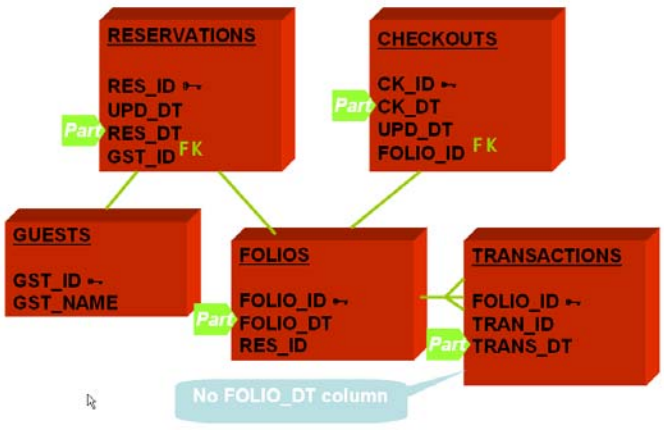


Figure 1 Design: 1st Pass

Design: 1st Pass

Now that we got all the answers, we get down to the design. Fig 1 shows the first pass of our design of the tables. The Primary Keys are shown by key icons, foreign keys by FK and partitioning keys are shown by the Part icon before the column name. The portioning keys are placed based on our initial design.

Design: 2nd Pass

The first pass assumes we partition month-wise. There is a huge problem. The TRANSACTIONS table, which has a many-to-one relationship with FOLIOS table, has a different partitioning key – TRANS_DT – than its parent – FOLIO_DT. There is no FOLIO_DT column in the child table. So, when you join the table, which happens all the time, you can't really take advantage of partition-wise joins. So, what can you do?

The easiest thing to do is to add a column called FOLIO_DT in the TRANSACTION table. Note, this completely goes against the principles of normalization – recording data at only one place. But this is an example of where puritan design has to meet the reality head on and you have to make decisions beyond text book definitions of modeling. Fig 2 shows the modified design after the second pass.

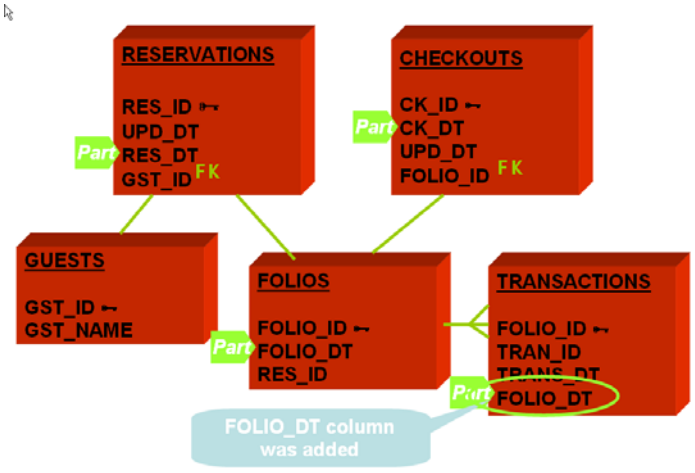
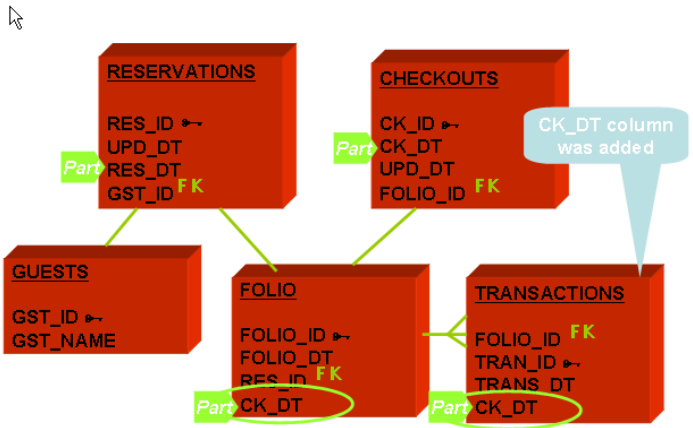


Figure 3 Design: 2nd Pass



Design: 3rd Pass

This solved the partition-wise join problem; but not others. Purge on CHECKOUTS, FOLIOS and TRANSACTIONS is based on CK_DT, not FOLIO_DT. FOLIO_DT is the date of creation of the record; CK_DT is updated at checkout. The difference could be months; so, purging can't be done on FOLIO_DT. We violated our first priority objective – data purge.

So, we come up with a solution: make CK_DT the Partitioning Key, since that will be used to purge. This brought up another problem – the column CK_DT is not present in all the tables. Well, we have a solution as well: add CK_DT to other tables. Again, you saw how we tweaked the model to accomplish our partitioning objectives. After adding the column, we made that column the partitioning key. Fig 3 shows the design after the third pass of the design process.

This was a key development in the design. Since the column CK_DT was on all the tables except RESERVATIONS, we can purge the tables in exactly same way.

Figure 2 Design 3rd Pass

Design: 4th Pass

While we solved the purging issue, we discovered some more issues as a result of the tweaked design.

(1) The records of the table FOLIOS are created at

check-in but the column CK_DT is updated at check-out. Since the column value could change, the record in FOLIOS may move to a different partition as a result of the update.

(2) The column CK_DT will not be known at check-in; so the value will be NULL. This will make it go to the PMAX partition. Later when the record is updated, the record will move to the correct partition.

The second problem is hard to ignore. It implies that *all* the records of the tables will always move, since the guests will checkout some day and the updates to the column will force row migration. The first problem is manifestation of the second; so if we solve the second, the first will automatically disappear.

So, we made a decision to make CK_DT NOT NULL; instead it is set to tentative date. Since we know how many nights the guest will stay, we can calculate the tentative checkout date and we will populate that value in the CK_DT. Again, we made a step against puritanical design principles in favor of real life solutions.
Our list of problems still has some entries. The TRANSACTIONS table may potentially have many rows; so updating CK_DT may impact negatively. Also, updating the CK_DT later may move a lot of rows across partitions; affecting performance even more. So, it may not be a good idea to introduce CK_DT in the TRANSACTION table.

So, we made a decision to undo the decision we earlier; we removed CK_DT from TRANSACTIONS. Rather we partition on the TRANS_DT, as we decided earlier. For purging, we did some thinking. The TRANS_DT column value will always be less than or equal to the CK_DT, since there will be no transactions after the guest checks out. So, even though the the partitioning columns are different, we can safely say that when a partition is ready for dropping in FOLIOS, it will be ready in TRANSACTIONS as well. This works out well for us. This also leaves no room for row migrations across partitions. Fig 4 shows the design after the 4th pass.

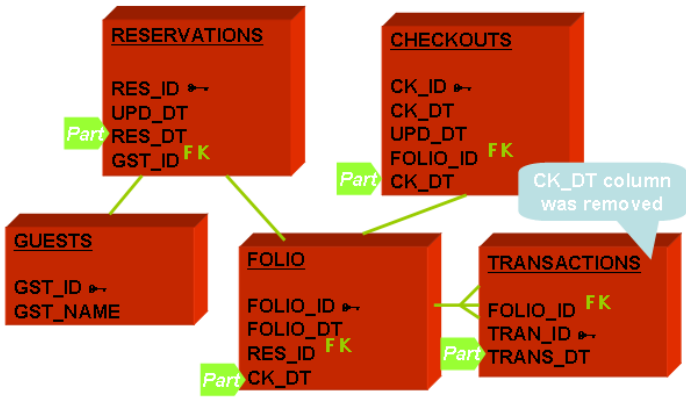


Figure 4 Design: 4th Pass

Scenario Analysis

One of the most important aspects of designing, including partitioning is thinking of several scenarios and how the design will hold up on each. Here we see different scenarios. The icons convey different meanings. I means a new row was created, U means the row was updated and M means the row was migrated to a different partition.

Scenario #1

Guest makes a reservation on Aug 31st for Sep 30th for one night, so checking out tentatively on Oct 1st. Every table has an update date column (UPD_DT) that shows the date of update. He actually checks out on Oct 2nd.

Records Created:

Table	Part Key	UPD_DT	Partition	
RESERVATIONS	09/30	08/31	Y08M09	I
Guest checks in on 9/30				
FOLIOS	10/01	09/30	Y08M10	I
Checks out on Oct 2nd:				
CHECKOUTS	10/02	10/02	Y08M10	I
TRANSACTIONS	10/02	10/02	Y08M10	I
FOLIOS	10/02	10/02	Y08M10	U

As you can see, all the records were created new. The only record to ever be updated is that of FOLIOS. But the record is not migrated from one partition to another.

Design: 5th Pass

While mulling over the design, we had a new thought: why not partition RESERVATIONS table by CK_DT as well? This action will make all the tables partitioned by the same column and the same way – the perfect nirvana for purging. When a guest checks out the reservations records are meaningless anyway. They can be queried with the same probability of the checkouts and folios; so it will be a boon for ILM and backup. Partition-wise joins will be super efficient, partition pruning between tables become a real possibility; and, most important of all, purging of tables will become much easier since we just have to drop one partition from each of the tables. So, we reached a decision to add a column CK_DT to the RESERVATIONS table and partition on that column. The new design is shown in Fig 5.

Scenario Analysis

Let's subject our design to some scenarios. First, let's see how the Scenario #1 holds up in this new design. The guest makes reservation on Aug 31st

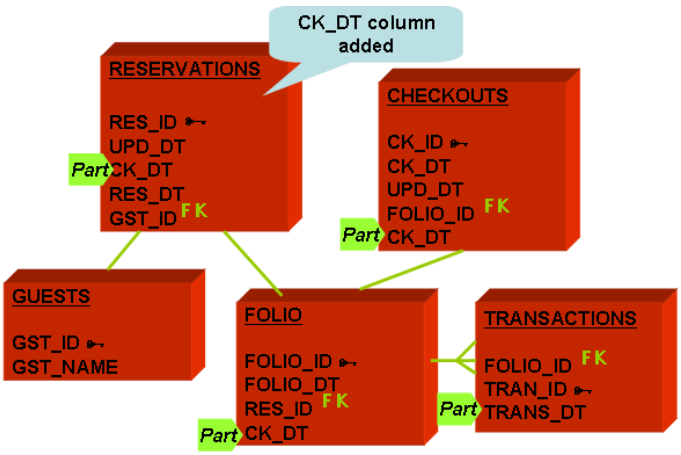


Figure 5 Design: 5th Pass

for one night on Sep 30th; so checking out tentatively on Oct 1st. However, instead of checking out on the intended day, he decided to stay one more day and checks out on Oct 2nd.

Records Created:

Table	Part Key	UPD_DT	Partition	
RESERVATIONS	10/01	08/31	Y08M10	I

Guest checks in on 9/30

FOLIOS	10/01	09/30	Y08M10	I

Checks out on Oct 2nd:

CHECKOUTS	10/02	10/02	Y08M10	I
TRANSACTIONS	10/02	10/02	Y08M10	I
RESERVATIONS	10/02	10/02	Y08M10	U
FOLIOS	10/02	10/02	Y08M10	U

This shows that two tables will be updated; but there will be no migration across partition boundaries – so far so good.

Scenario #2

It's a modification of the Scenario #1. the guest checks out on Nov 1st, instead of Oct 1st.

Records Created:

Table	Part Key	UPD_DT	Partition	
RESERVATIONS	10/01	08/31	Y08M10	I

Guest checks in on 9/30

FOLIOS	10/01	09/30	Y08M10	I

Checks out on Nov 1st:

CHECKOUTS	11/01	11/01	Y08M11	I
TRANSACTIONS	11/01	11/01	Y08M11	I
RESERVATIONS	11/01	11/01	Y08M10	M
FOLIOS	11/01	11/01	Y08M10	M

Consider the case carefully. The design reeks of two bad ideas in partitioning – row migration; but how prevalent is it? If you examine the scenario, you will notice that the only case the row migration will occur is when rows change months. When checkout date was changed from 10/1 to 10/2, the record was updated; but the row didn't have to move as it was still in the Oct 08 partition. The row migration occurred in the second case where the month changed from October to November. How many times does that happen? Perhaps not too many; so this design is quite viable.

Here you saw an example of how an iterative design approach was employed to get the best model for partitioning. In the process, we challenged some of the well established rules of relational design and

made modifications to the logical design. This is all perfectly acceptable in a real life scenario and is vital for a resilient and effective design.

New Column for Partitioning

In the design, we added a column CK_DT to many tables. How do we populate it? There are two sources for populating it – applications and triggers. If the design is new and the coding has not begun, the apps can easily do it and in many cases preferred as it is guaranteed. If this is an established app, then it has to be modified to place the logic. In that case, the trigger approach may be easier.

Non-Range Cases

So far we have discussed only range partitioning cases. Let's consider some other cases as well. Consider the GUESTS table, which is somewhat different. It has:

- 500 million+ records
- No purge requirement
- No logical grouping of data. GUEST_ID is just a meaningless number
- All dependent tables are accessed concurrently, e.g. GUESTS and ADDRESSES are joined by GUEST_ID

So, No meaningful range partitions are possible for this table. This is a candidate for hash partitions, on GUEST_ID. We choose the number of partitions in such a way that each partition holds about 2 million records. The number of partitions must be a power of 2. So, we chose 256 as the number of partition.

All dependent tables like ADDRESSES were also hash partitioned on (guest_id), same as the GUESTS table. This type of partitioning allows great flexibility in maintenance.

Hotels Tables

The table HOTELS holds the names of the hotels Several dependent tables – DESCRIPTIONS, AMENITIES, etc. – are all joined to HOTELS by HOTEL_ID column. Since HOTEL_ID varies from 1 to 500, could this be a candidate for Partitioning by LIST?

To answer the question, let's see the requirements for these tables. These are:

- Very small
- Do not have any regular purging need
- Mostly static; akin to reference data
- Not to be made read only; since programs update them regularly.

So, we took a decision: not to partition these tables.

Tablespace Decisions

The partitions of a table can go to either individual tablespaces or all to the same tablespace. How do you decide what option to choose?

Too many tablespaces means too many datafiles, which will result in longer checkpoints. On the other hand, the individual tablespaces option has other benefits.

- It affords the flexibility of the tablespaces being named in line with partitions, e.g. tablespace RES0809 holds partition Y08M09 of RESERVATIONS table. This makes it easy to make the tablespace READ ONLY, as soon as you know the partition data will not be changed.
- Easy to backup – backup only once, since data will not change
- Easy to ILM, since you know the partitions
- Allows the datafiles to be moved to lower cost disks

```
ALTER DATABASE DATAFILE '/high_cost/…'
RENAME TO '/low_cost/…';
```

Neither is a perfect solution. So, we proposed a middle of the way solution. We created a tablespace for each period, e.g. TS0809 for Sep '08. This tablespace contains partition Y08M09 for all the tables – RESERVATIONS, CHECKOUTS, TRANSACTIONS and soon. This reduced the number of tablespaces considerably.

Partitions of the same period for all the tables are usually marked read only. This makes the possible

to make a tablespace read only, which helps backup, ILM and other objectives. If this conjecture that the tablespace can be read only is not true, then this approach will fail.

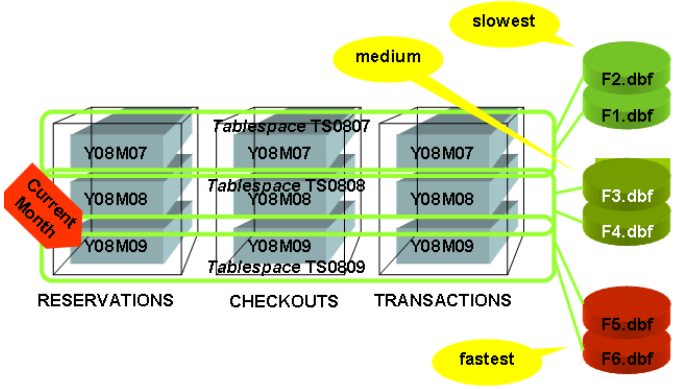


Figure 6 Tablespace Design

Figure 6 shows the final tablespace design. Here we have defined just three tablespaces TS0807, TS0808 and TS0809. The tables – RESERVATIONS, CHECKOUTS and TRANSACTIONS – have been partitioned exactly in the same manner – monthly partitions on some date. The partitions are named Y08M07, Y08M08 and Y08M09 for July, August and September data respectively. All these partitions of a particular period for all tables go to the corresponding tablespaces. For instance, tablespace TS0809 holds the RESERVATION table's partition Y08M09, CHECKOUTS table's partition Y08M09 and TRANSACTION table's partition Y08M09. Suppose the current month is Sep 08. this means that the files for tablespace TS0809 will be on the fastest disk; TS0808 will be on medium disk and the third one will be on the slowest disk. This will save substantially on the storage cost.

Summary

Partitioning Tips

1. Understand clearly all benefits and use cases of partitioning, especially the ones that will or will not apply in your specific case.
2. List the objectives of your design – why you are partitioning – in the order of priority.
3. If possible design the same partitioning scheme for all related tables, which helps purging, ILM, backup objectives.
4. To accomplish the above objective, don't hesitate to introduce new columns
5. Try to make all indexes local, i.e. partition key is part of the index. This help management easier and the database more available.

Tips for Choosing Part Key

1. If a column is updateable, it does not automatically mean it is not good for partitioning.
2. If partition ranges are wide enough, row movement across partitions is less likely
3. Row movement may not be that terrible, compared to the benefits of partitioning in general; so don't discard a partitioning scheme just because row movement is possible

Oracle 11g Enhancements

Oracle Database 11g introduced many enhancements in the area of partitioning. Of several enhancements, two stand out as particularly worthy from a design perspective.

Virtual Column Partitioning

It allows you to define partitioning on a column that is virtual to the table, i.e. it is not stored in the table itself. Its value is computed every time it is accessed. You can define this column as a partitioning column. This could be very useful in some cases where a good partitioning column does not mean you have to make a schema modification.

Reference Partitioning

This feature allows you to define the same partitioning strategy on the child tables as the parent table, even if the columns are not present. For instance, in the case study you had to add the CK_DT to all the tables. In 11g, you didn't have to. By defining a range partitioning as "by reference", you allow Oracle to create the same partitions on a table as they are on parent table. This avoids unnecessary schema changes.

Thank you from Proligence.