1 Partitioning

The basic idea of partitioning is letting the data itself determine the location where it is stored—as well as what indexes apply where.

There are various reasons to do this. For example, if you manage a data warehouse, and all (or most) of your data can logically be thought of separated into days (or weeks, months, etc.) then you can define a partition by day: store all the data for the day in the same place.

With most partitioning schemes, indexes are localized per partition. You can drop or rebuild an index just for the partition that is being updated. Accessing partitions is generally independent from other partitions (you can access partition1 while partition2 is being loaded).

2 Do-it-yourself Partitioning

You can do 'fake' partitioning with the help of a 'union all' operator. You create many identical tables, such as:

```
create table SOMEDATA_20080310 (
    rdate date,
    blah varchar(10),
    glah int
);
create table SOMEDATA_20080311 (
    rdate date,
    blah varchar(10),
    glah int
);
```

Note that the tables have identical DDL, and the only difference is something in the table name. We ensure to store '20080310' data into '`SOMEDATA_20080310`' and '20080311' data into '`SOMEDATA_20080311`', etc., In other words, we would have 365 tables for a year, etc. Note that all indexes are per table (they don't span days).

What we do next is define a view that is a union all of all of these tables, ie:

```
create view SOMEDATA as
select * from SOMEDATA_20080310
union all
select * from SOMEDATA_20080311
union all
select * from SOMEDATA_20080312
union all
select * from SOMEDATA_20080313
union all
select * from SOMEDATA_20080314
```

Obviously if we had 1000 such tables, we would 'union all' all of them.

Whenever we need data that we're certain is in some table, we can select it directly from the partition, ie:

```
select *
from SOMEDATA_20080310
where glah > 42;
```

Anytime we're not sure where the data is, we may simply select from the `SOMEDATA` view, which will hit all tables, ie:

```
select *
from SOMEDATA
where rdate between '20080310' and '20080314' and glah > 42
```

Note that if database is smart enough, the previous query will be relatively efficient. For example, if each table has an index defined on `rdate`, then all it has to do per table is hit the index, and quickly determine if the date (in that range) is present in the table. In other words, if the above query is slow, find out why, and there are usually ways of fixing it (either by defining an index or contacting the database vendor and reporting this as a bug).

2.1 Other thoughts

In this do-it-yourself scheme, you're responsible for ensuring where data actually ends up. If you load '20080310' data into the the wrong table, that's your fault. You should probably define some constraint that prevents you from having such a situation.

Another issue is that everytime you add data (normally every day or so), you will need to re-create the view. The same applies for dropping previous days.

3 Oracle Partitioning

Oracle partitioning is somewhat similar to the do-it-yourself scheme above, except you don't need to mess with views, nor worry about data falling into the wrong partition.

You can create a 'partitioned' table via:

```
create table SOMEDATA (
    rdate date,
    blah varchar(10),
    glah int
)
PARTITION BY RANGE (rdate) (
    PARTITION PPREVDATA VALUES LESS THAN (TO_DATE('20080310','YYYYMMDD')),
    PARTITION P20080310 VALUES LESS THAN (TO_DATE('20080311','YYYYMMDD')),
    PARTITION P20080311 VALUES LESS THAN (TO_DATE('20080312','YYYYMMDD')),
    PARTITION P20080312 VALUES LESS THAN (TO_DATE('20080313','YYYYMMDD')),
    PARTITION P20080313 VALUES LESS THAN (TO_DATE('20080314','YYYYMMDD')),
    PARTITION P20080314 VALUES LESS THAN (TO_DATE('20080315','YYYYMMDD'))
);
```

Whenever you load (sqlldr) or insert data, it will land on the appropriate partition. You can then select the data as you normally would, ie:

```
select *
from SOMEDATA
where glah < 42
```

Or if you're absolutely sure of which partition you want to go after, you can select just from that partition, which is generally much faster, as it's only going after a small part of the whole table:

```
select *
from SOMEDATA partition (P20080312)
where glah < 42
```

Note that the above is equivalent to:

```
select *
from SOMEDATA
where rdate between
          TO_DATE('20080312','YYYYMMDD') and
          TO_DATE('20080313','YYYYMMDD')
    and glah < 42
```

To add partitions (as you'd need to do on a regular basis, as you load data), you can do:

```
ALTER TABLE SOMEDATA
ADD PARTITION P20080315
     VALUES LESS THAN (TO_DATE('20080316','YYYYMMDD'))
```

Working with partitions is relatively flexible. You can do just about all database operations on a single partition. You can also move partitions between different table spaces (move the most active partition to the fastest disk, etc.)

In some cases, you might also want to maintain the 'next' partition, and then split it to create daily partitions (as opposed to adding a new partition).

3.1 Other Thoughts

Oracle has different partition types. The one described above is the 'range' partition, and is likely the most common one used. Others include hash (data is distributed on some hash code derived from the data—useful for load balancing, but useless for data ranges), list (instead of a range, you specify a list of values—for example, if you distribute by 'state', then all the data for 'NY' will fall into the same partition).

If your database is growing beyond some 'reasonable' size (1gig of data?), you should be using some partitioning strategy. Indexes will only get you so far (it's much easier to work with many partitions, to load, re-index, etc., individual sections of tables, as opposed to one huge table).

4 PostgreSQL Partitioning

PostgreSQL handles partitioning via inheritence. The way this works is: You define a 'master' table that will store no data, ie:

```
create table SOMEDATA (
     rdate date,
     blah varchar(10),
     glah int
)
```

You then define a bunch of other 'child' tables that inherit from the parent, except that have a 'check', to ensure they only contain the right data, ie:

```
create table SOMEDATA_20080310 (
    CHECK ( rdate >= DATE '2008-03-10' AND rdate < DATE '2008-03-11' )
) INHERITS (SOMEDATA);
create table SOMEDATA_20080311 (
    CHECK ( rdate >= DATE '2008-03-11' AND rdate < DATE '2008-03-12' )
) INHERITS (SOMEDATA);
create table SOMEDATA_20080312 (
    CHECK ( rdate >= DATE '2008-03-12' AND rdate < DATE '2008-03-13' )
) INHERITS (SOMEDATA);
```

You can define as many of these 'child' tables as you want. You should then create an index on the partition column, ie:

```
CREATE INDEX SOMEDATA_20080310_rdate ON SOMEDATA_20080310 (rdate);
CREATE INDEX SOMEDATA_20080311_rdate ON SOMEDATA_20080311 (rdate);
CREATE INDEX SOMEDATA_20080312_rdate ON SOMEDATA_20080312 (rdate);
```

For inserting data, someone will need to be aware that they must insert data into the 'child' tables as opposed to the parent table.

```
INSERT INTO SOMEDATA_20080310 VALUES (DATE '2008-03-10','meh',42);
```

Once the data is in there, you can select data using the master table, ie:

```
select * from SOMEDATA where blah='meh';
```

There is a way to allow users to insert data into a specific partition, by creating a rule:

```
CREATE OR REPLACE RULE SOMEDATA_last AS
ON INSERT TO SOMEDATA
DO INSTEAD
    INSERT INTO SOMEDATA_20080310
    VALUES ( NEW.rdate, NEW.blah, NEW.glah );
```

The above will allow someone to insert data into 'SOMEDATA' and have it fall into `SOMEDATA_20080310` child table. This rule will need to be modified often, etc., and is generally a bad way of solving this problem (ie: ensuring users insert and load data appropriately is generally preferred).

We can expand the 'last' partition approach to pick one among many partitions, by specifying a where clause in the rule, ie:

```
CREATE OR REPLACE RULE SOMEDATA_20080310_rule AS
ON INSERT TO SOMEDATA
     WHERE (rdate >= DATE '2008-03-10' AND rdate < DATE '20080311')
DO INSTEAD
    INSERT INTO SOMEDATA_20080310
    VALUES ( NEW.rdate , NEW.blah , NEW.glah );

CREATE OR REPLACE RULE SOMEDATA_20080311_rule AS
ON INSERT TO SOMEDATA
     WHERE (rdate >= DATE '2008-03-11' AND rdate < DATE '20080312')
DO INSTEAD
    INSERT INTO SOMEDATA_20080311
    VALUES ( NEW.rdate , NEW.blah , NEW.glah );
```

Obviously we would need many such rules. But then to have the data land in the right partition, we would simply need to do:

```
INSERT INTO SOMEDATA VALUES (DATE '2008-03-10','meh',42);
```

4.1 Other thoughts

Inheritence is a relatively clean way of doing partitioning. You can control inheritence state of a table:

```
ALTER TABLE SOMEDATA_20080310 NO INHERIT SOMEDATA;
```

Suddenly, `SOMEDATA_20080310` data will no longer show up when you query `SOMEDATA`. You can even drop it, etc.

This is useful for setting up tables (partitions), loading them, etc., and then with one instruction making them 'visible' to the world.

5 SQL Server Partitioning

Google (or msn.com, eh) for it.

6 Netezza, Teradata, and others

There are databases that take a slightly different approach when it comes to partitions. Instead of locating data in the same place (to ensure table scans don't traverse everything, etc.) they *distribute* the data based on the partition keys. In those cases, the partition key is known as a distribution key, and the idea is to get them to distribute the data over as many disks (or machines) as possible. For example (Netezza):

```
create table SOMEDATA (
     rdate date,
     blah varchar(10),
     glah int
) distribute on random;
```

will randomly distribute the data. Your database instance could have 200 (or any number in fact) storage servers, and upon an insert your record will randomly land into any one (or several) of the boxes.

Performance is achieved through massive parallelization of table scans. Whenever you issue a:

```
select *
from somedata where glah > 42;
```

Your query is sent to all of the 200 (or more) storage servers. Each server does a table scan to get the result set. Some node aggregates the results into a single result set that's returned to the user. The whole scheme allows simple table-scan queries to be 200 times faster (and to scale linearly, so if you have 800 professors, then each query becomes 800 times faster, etc.)