ExtenDB Planning Guide

ExtenDB Parallel Server

Version 1.1

October 2006

Table of Contents

1 Introduction

1.1 Purpose

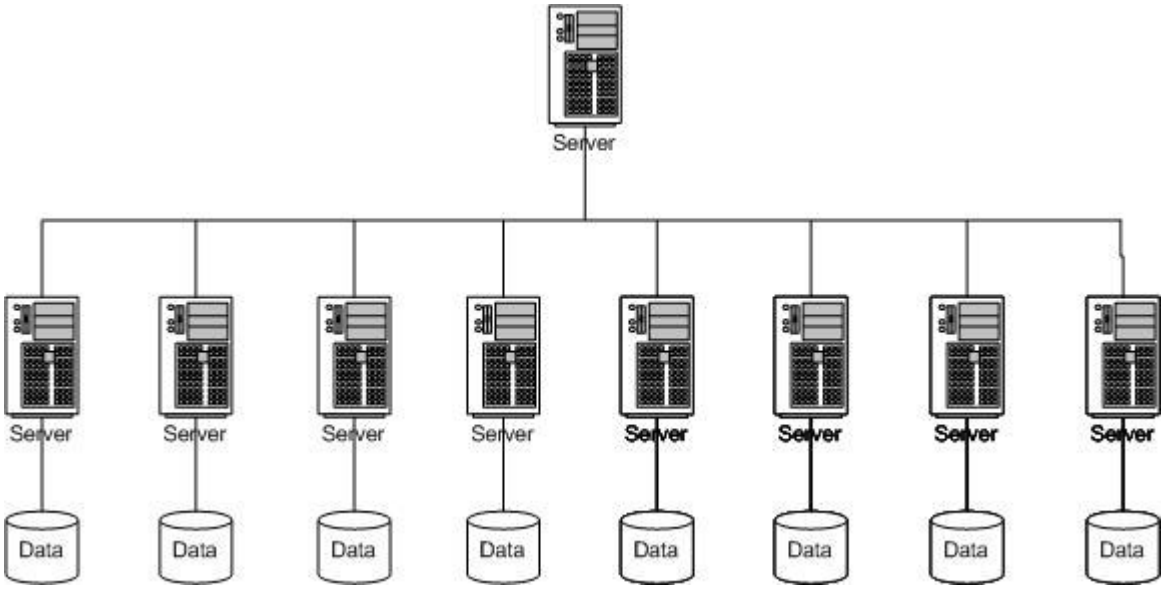
The purpose of this document is to provide information to customers of the ExtenDB Parallel Server to allow them to properly plan for its installation and configuration. To help achieve this, hardware purchasing considerations and database schema are discussed.

1.2 Overview

ExtenDB is a parallel, clustered database system designed for data warehousing solutions. Its objective is to allow queries that run against large volumes of data to execute quickly.

Our system runs on top of other systems, called nodes, each of which houses its own independent database. Each underlying database is an Open Source database like PostgreSQL, running in a shared-nothing architecture, typically using commodity PC hardware.

To allow for quick response times, the database administrator (DBA) employs strategies to partition each table in the database, based on a computed hash of a column. The optimizer intelligently recognizes when to join tables locally or when it needs to send data around. Various other techniques are used to achieve linear or near-linear scalability.



2 Platform

2.1 Operating System

You can use any operating system you choose, provided it supports the underlying database (PostgreSQL) as well as the Java Runtime Environment 5.0 or later. We have tested with Linux and Windows.

2.2 Environment

The ExtenDB Server is written in Java and communicates with underlying databases with JDBC. Java Runtime Environment 5.0 or later is required. User applications can connect to ExtenDB via the ExtenDB JDBC driver, ExtenDB ODBC driver, or perl DBI.

Although the ExtenDB Parallel Server runs in a Java Virtual Machine, it performs quite well. It should be pointed out that most of the resource intensive work is done within the native underlying databases, with the ExtenDB Server coordinating the work.

The ExtenDB Parallel Server Standard Edition has a single process that executes on the coordinator node. The other nodes merely have the chosen underlying database utilized, such as PostgreSQL.

The regular version of the ExtenDB Parallel Server differs from the four node Standard Edition in that it additionally has an ExtenDB Agent process running on each node to facilitate greater scalability.

2.3 Hardware

Typically PC-based servers are used, due to cost. More detailed information appears in the Hardware Considerations section.

3 Hardware Considerations

3.1 Introduction

In choosing ExtenDB, you were likely attracted to the price/performance value proposition that it offered. This is achieved in part due to using commoditized PC-based hardware. A discussion of what to consider when purchasing hardware appears below, along with some recommendations.

3.2 Nodes

Within a node there are various components that influence the total performance of the system, in particular, the CPU, memory, the bus speed, networking components, and (hard disk) storage.

Note the order of the list just stated is in approximate order of the fastest to slowest components. Hard disks are electro-mechanical devices with moving parts and are much slower than CPUs of course.

We also want to avoid making any single component a bottleneck and achieve some balance on each system, but while doing so at a reasonable cost.

3.2.1 Storage

The nodes in the ExtenDB cluster will need to read from disk often. To achieve good performance, you should consider using RAID, such as RAID 10 (striping and mirroring) or RAID-5 (striping with parity), with a good disk controller. This allows you to get multiple drives working at the same time. If the SCSI interface is used, you may want to get a controller card with multiple channels to achieve greater data transfer rates from the drives. Depending on your budget, you may want to consider using SATA instead of SCSI, which offers a good value.

In addition to speed gains through striping, the other advantage of using RAID is redundancy. In a RAID-5 configuration, you can achieve both greater throughput through multiple drives, while allowing the node to be able to continue working if there is a drive failure. RAID 10 (1+0) has the best speed advantage with redundancy (although at the penalty of less space utilization). With many nodes and drives in your ExtenDB cluster, you will likely have a drive failure at some point, so using RAID should be considered a necessity.

Another option is to use software RAID, which appears to be gaining more proponents recently, given the ever-faster processors available.

The drives themselves should have a reasonably fast seek time. You may consider purchasing drives with a high RPM rate.

You should have the node use directly attached storage. If external storage must be used, to avoid thrashing and achieve parallelization, it is recommended that a set of physical drives are assigned to a node in a dedicated manner, as opposed to logically sharing some drives amongst the nodes.

You should consider having separate devices for temp space, apart from where table data is stored. ExtenDB allows you to define tablespaces, where a logical tablespace group maps to tablespaces on each of the nodes. You can also use this in determining placement of your tables as well.

3.2.1.1 Storage Requirements

When deciding on how many individual nodes you want with how many hard disks for each, you should have a good idea of the size of your database. Take into account any indexes that you may create. If you replicate tables or denormalize ("pre-join") data, you will need more space (more information on denormalization appears in the Database Schema section). Finally, you should have a lot of extra space for temp tables. ExtenDB uses temp tables to store intermediate results as queries are processed. It is recommended to not use more than 60% of the available database storage on a node, but this will vary depending on your schema and queries.

3.2.2 Networking / Interconnectivity

A lot of data will be moving around amongst the nodes, so high-speed connectivity is required between them. Gigabit Ethernet offers the best value in accomplishing this. There are other alternatives such as fibre channel, but it is more expensive.

3.2.3 Bus

Many systems still use PCI as the peripheral bus. You may look for systems that use PCI-X, which allows for faster speeds. It does require you to purchase more expensive PCI-X cards however.

3.2.4 Memory

One of the most important ways to get more performance out of a system is to have ample memory. More memory allows the database to use more cache. More cache means more of the database can be stored in memory, reducing disk accesses. A caveat here is that for large volumes of data and data warehousing queries that need to scan entire tables, cache rates may be low. Nonetheless, ample memory is a

necessity, and helps out tremendously with internal processing and intermediate results.

When purchasing your systems, get as much memory as you can within your budget. Keep in mind that it is not just the underlying tables in the database that will use it. Temporary tables will be created and more memory will be available for them as well means less swapping to disk.

3.2.5 CPU

When deciding on a CPU, it is usually best to look for the best performing one at the most reasonable cost. Very often the very fastest available will only offer minor improvements in speed while costing significantly more than a processor slightly slower. You may be better off spending your hardware budget elsewhere.

What underlying database you are using may influence your decision on whether to use dual processor systems. If using PostgreSQL, it should be pointed out that a single query can only use a single CPU, but if you have additional queries running, they will be able to take advantage of the second CPU. In any event, a dual CPU system is a good idea for the coordinator.

3.3 The Coordinator

You will designate one of the nodes as the coordinator. This is also typically where metadata information is stored. The load will be a little heavier on the coordinator, so it is a good idea to consider getting extra memory and perhaps a faster processor on the coordinator. Here you may also want to consider a dual processor system.

3.4 Networking / Interconnectivity

As mentioned above for the nodes, Gigabit Ethernet over copper offers the best value. If you choose that, accordingly, you will want to purchase Gigabit Ethernet switches.

You may want to consider configuring the nodes in the cluster on their own subnet, for better security, while making the non-coordinator nodes accessible only to the coordinator.

4 Database Schema

4.1 Introduction

One of the most critical factors in designing your decision support database is the database schema you chose. Due to the nature of how the ExtenDB Server works, we want to allow for as much parallelization as possible. Strategies for achieving this appear below.

Our examples use tables from the TPC-H test schema to better demonstrate various options, with lineitem acting as a fact table.

4.2 Tables

4.2.1 Replicating Tables

Replicating in this context refers to creating an exact duplicate of a table on all of the nodes. We want to do this for smaller lookup tables, like a state_code table containing all of the states in the United States. It would not make much sense to partition a table like this that just has 50 rows. We will spend a lot of time unnecessarily copying them around. It is best to just keep a copy of the table on all nodes. We can do this by using the REPLICATED clause in the CREATE TABLE statement.

Example:

```
CREATE TABLE region  (
r_regionkey  integer not null,
r_name       char(25) not null,
r_comment    varchar(152)) REPLICATED
```

Now, when region is joined with another table in the query, the join can take place locally on each node in parallel immediately, without having to worry about shipping rows around.

4.2.2 Internode Partitioning

In order to achieve fast query times involving large tables, we want to keep all of the nodes busy working while a query is executing. To do that, we want to distribute data in a given table across all of the nodes.

To best explain how this works, some examples are used.

With the CREATE TABLE statement, the DBA can specify how he or she wants to partition the table. This consists of specifying a column and on which nodes to distribute the data. For example:

```
CREATE TABLE orders  (
o_orderkey       INTEGER NOT NULL,
o_custkey        INTEGER NOT NULL,
o_orderstatus    CHAR(1) NOT NULL,
o_totalprice     DECIMAL(15,2) NOT NULL,
o_orderdate      DATE NOT NULL,
o_orderpriority  CHAR(15) NOT NULL,
o_clerk          CHAR(15) NOT NULL,
o_shippriority   INTEGER NOT NULL,
o_comment        VARCHAR(79) NOT NULL) PARTITIONING KEY o_orderkey
ON ALL
```

"ON ALL" here indicates that all nodes defined for the available cluster should be used. We could have also specified a subset of nodes by giving their node id numbers as well. In practice, it makes sense to utilize all of the available nodes, unless you are using a dedicated coordinator and wish to keep partitioned data off of it.

The partitioning key in the example is o_orderkey. A hash will be calculated based on o_orderkey's value for each row and will be mapped to one of the nodes to determine where the row will reside. An even distribution across all nodes will be achieved.

Now, if we were to execute a singleton SELECT such as

```
SELECT * FROM orders WHERE o_orderkey = 1000
```

the ExtenDB server will recognize that o_orderkey is the partitioning key, and will calculate a hash value to determine which node is responsible for value 1000. It will then query only the single node.

In your decision support environment, you will more likely be executing larger queries, like

```
SELECT MAX(o_totalprice)
FROM orders
WHERE o_orderdate >= '01/01/2003'
```

This type of query scales very well. In this case, the ExtenDB Server will execute that exact same query on each of the nodes. In a 16 node system, it will receive 16 different results, each of which is the MAX(o_totalprice) that is on the particular node. It will then take the MAX value of those 16 results, to arrive at the final MAX result.

4.2.3 Constraint Exclusion Partitioning

PostgreSQL and Bizgres Only

Note: this is different from ExtenDB's internode partitioning.

PostgreSQL 8.1 and Bizgres 0.7 introduce the notion of "partitioning" within a database instance via check constraints. This allows the DBA to create segments for a table that contain ranges of values, for example. A table named orders could be partitioned into monthly subtables, allowing queries that include a condition based on order date to scan with a smaller set of data, and therefore have a faster query time.

This is a powerful feature that should be taken advantage of. Constraint exclusion partitioning coupled with ExtenDB's partitioning across multiple nodes will result in significantly faster query response times; a large table can be broken into multiple subtables, each of which is partitioned across multiple nodes in the ExtenDB cluster.

An example appears below.

```
CREATE TABLE orders  (
 o_orderkey      INTEGER NOT NULL,
 o_custkey        INTEGER NOT NULL,
 o_orderstatus    CHAR(1) NOT NULL,
 o_totalprice     DECIMAL(15,2) NOT NULL,
 o_orderdate      DATE NOT NULL,
 o_orderpriority  CHAR(15) NOT NULL,
 o_clerk          CHAR(15) NOT NULL,
 o_shippriority   INTEGER NOT NULL,
 o_comment        VARCHAR(79) NOT NULL)
PARTITIONING KEY o_orderkey ON ALL;

CREATE TABLE orders_199201
( CHECK (o_orderdate BETWEEN '19920101'::DATE AND '19920131'::DATE) )
INHERITS (orders);

CREATE TABLE orders_199202
( CHECK (o_orderdate BETWEEN '19920201'::DATE AND '19920228'::DATE) )
INHERITS (orders);

:
```

Any query like "SELECT count(*) from orders where o_orderdate between '1992-01-01' and '1992-01-15'" will only use tuples found from the orders_199201 subtable (and the orders table, which should be left empty).

Note that when loading data, you must insert data into the proper subtable. Using the above example, in the current implementation, loading into orders directly will not automatically just insert the data into the correct subtable.

Another important consideration when creating subtables constraints is to be aware that the current PostgreSQL implementation is a bit datatype sensitive, and you

might find that the PostgreSQL executor is not taking full advantage of eliminating subtables.
In the case of dates, we recommend using the above syntax to cast it a date type, as in

```
CHECK (o_orderdate BETWEEN '19920101'::DATE AND '19920131'::DATE)
```

Leaving it as either just a date, or as a quoted string may cause queries in PostgreSQL to not be executed optimally. This depends on how the CHECK constraints are formulated and how the WHERE conditions are formulated. The above check constraint syntax appears to handle various date constructs (quoted, cast) in SELECT WHERE clauses properly.

4.2.4 How Joins Affect Partition & Replication Strategy

Now that we have covered some of the basics, some more advanced examples are discussed.

Assume we have the following table, lineitem, that acts as a fact table in our database:

```
CREATE TABLE lineitem (
l_lineitemkey INTEGER NOT NULL,
l_orderkey    INTEGER NOT NULL,
l_partkey     INTEGER NOT NULL,
l_suppkey     INTEGER NOT NULL,
l_linenumber  INTEGER NOT NULL,
l_quantity    DECIMAL(15,2) NOT NULL,
l_extendedprice  DECIMAL(15,2) NOT NULL,
l_discount    DECIMAL(15,2) NOT NULL,
l_tax         DECIMAL(15,2) NOT NULL,
l_returnflag  CHAR(1) NOT NULL,
l_linestatus  CHAR(1) NOT NULL,
l_shipdate    DATE NOT NULL,
l_commitdate  DATE NOT NULL,
l_receiptdate dATE NOT NULL,
l_shipinstruct CHAR(25) NOT NULL,
l_shipmode     CHAR(10) NOT NULL,
l_comment      VARCHAR(44) NOT NULL) PARTITIONING KEY l_lineitemkey
ON ALL
```

We want to execute the following query:

```
SELECT
        l_orderkey,
        SUM(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
```

```
FROM
        orders,
        lineitem
WHERE
         l_orderkey = o_orderkey
    AND l_shipdate > '1994-09-20'
GROUP BY
        l_orderkey,
        o_orderdate,
        o_shippriority
ORDER BY
        revenue desc,
        o_orderdate;
```

Note that a join occurs on lineitem.l_orderkey = orders.o_orderkey. Assume that in looking at our table definitions, the table orders was partitioned on o_orderkey, and lineitem was partitioned on l_lineitemkey. It could be that a given o_orderkey value is on node 7 and needs to join with the equivalent l_orderkeys on nodes 4, 9 and 13. Since they are on different nodes, the ExtenDB server will need to ship data around for joining. While we take advantage of all the nodes we have and it will execute fine, it will not be as quite as fast as if the needed rows were on the same node.

Now assume that l_lineitem was instead partitioned on l_orderkey. In this case, the ExtenDB server will recognize that the join condition occurs on expressions that match the partitioning scheme of the tables involved. It will therefore perform the join locally on each node, instead of needing to ship rows.

This example illustrates the importance of choosing good partitioning keys. Although the database may be used in environments where ad-hoc querying tools are available, the DBA should try and choose sensible partitioning keys for joins that are likely to occur.

4.2.4.1 Another Example

We will examine another more complex example. It is somewhat contrived, as well as with some possible alternatives discussed, just to illustrate things to consider when determining how to determine your schema and partitioning.

```
SELECT
 c_custkey,
 c_name,
 sum(l_extendedprice * (1 - l_discount)) as revenue,
 c_acctbal,
 n_name,
 c_address,
 c_phone,
 c_comment
FROM
```

```
 customer,
 orders,
 lineitem,
 nation
WHERE
 c_custkey = o_custkey
 AND l_orderkey = o_orderkey
 AND o_orderdate >= '1994-07-01'
 AND o_orderdate < '1994-07-02'
 AND l_returnflag = 'R'
 AND c_nationkey = n_nationkey
GROUP BY
 c_custkey,
 c_name,
 c_acctbal,
 c_phone,
 n_name,
 c_address,
 c_comment
ORDER BY revenue desc;
```

Assume that nation is REPLICATED on all of the nodes here, with customer partitioned on c_custkey, orders on o_orderkey and lineitem on l_orderkey. The ExtenDB Server will need to execute this query in a couple of steps. There are various alternatives that the Optimizer will evaluate, including:

- It could join customer and nation first, then join these results with orders and lineitem.
- It could join orders and lineitem first, then join these results with customer and nation.

The join pairs customer-nation and orders-lineitem can each take place at the individual nodes without row shipping occurring. The customer-nation join involves nation, a replicated look-up table, so no rows need to be shipped. The orders-lineitem join involves a parent-child join, so no rows need to be shipped here either.

In either case, there is an intermediate step of rows having to be shipped. The query will scale well, however. Consider the case where customer is replicated. If that were the case, we could perform a join on all 4 of the tables at once locally at the nodes and eliminate the additional step.

Replicating customer on all of the nodes will help in cases like this, but there are things to consider and trade-offs.

- The cardinality of orders to customer and the number of nodes. This may influence how close to linear the query can scale as more nodes are added.
- The other expected queries. If users perform a lot of queries only against the customer table without joining with others, it will be executed on just one node instead of others, and you lose parallelism. Also, joins will occur against

a bigger table on each node, whatever the join columns may be. In cases like this, it is probably not a good idea to replicate.

- Note that if there are a lot of queries just against this single customer table, another alternative would be for the DBA to set up two customer tables; one that is replicated and one that is partitioned. It is up to your load process to load up both tables. Also, reports that users run will have to choose the appropriate table.

We could have also set up our schema so that o_custkey is the partition column for orders. The drawback to that though is that lineitem can no longer be joined locally with orders on l_orderkey without shipping rows.

An alternative to that would be to add the additional foreign key "l_custkey" to lineitem, and partitioning by it, o_custkey for orders and c_custkey for customer. We need to ensure that joins between orders and lineitem appear as "orders.o_custkey = lineitem.l_custkey and orders.o_orderkey = lineitem.l_orderkey". By doing this, if customer is also joined with orders the usual way, we can join all 3 tables together locally on the nodes without any row shipping, and customer remains partitioned for faster parallelism against it.

To summarize, there are various considerations and trade-offs to weigh when choosing your schema. It is also best to be familiar with the types of queries that will be run against your database to guide decisions. Finally, the schema you decide on will influence your ETL process to ensure data is loaded up properly, perhaps denormalized as well, if desired.

4.3 Denormalization

It may also be advantageous to denormalize the data. Partially denormalizing a schema is a standard technique in building star schemas used in data warehousing.

Denormalization is the opposite of normalization. In an OLTP relational database, one normally tries to avoid any redundant data by normalizing the data. In data warehousing, it may be advantageous to do just the opposite, "pre-joining" the data in effect.

The cost of disk storage is cheap, so doing this will save a lot of work instead of doing joins later. It will also reduce the likelihood of row shipping occurring, depending on your schema.

An example of this is orders and lineitem, using an earlier example. Lineitem could be changed to also contain all of the columns in the order table. Queries can now just use this table instead of performing the orders and lineitem join each time.

Consider the following query:

```
SELECT
 n_name,
 SUM(o_ordertotal) as revenue
FROM
 customer,
 orders,
 nation,
 region
WHERE
 c_custkey = o_custkey
 AND o_orderdate >= '1994-01-01'
 AND o_orderdate < '1994-12-31'
 AND c_nationkey = n_nationkey
 AND r_name = 'ASIA'
 AND n_regionkey = r_regionkey
GROUP BY  n_name
ORDER BY revenue desc;
```

If it turns out that a lot of queries being executed are based on the country of the customer, it may make sense to add in the additional country or other customer information into the orders table.

Assume we add in the country of the customer in the orders table as o_cust_nation. Also, we could add in region information into the nation table like n_regionname. Now, our query is much simpler:

```
        SELECT
         n_name,
         SUM(o_ordertotal)
        FROM
         orders,
         nation
        WHERE
                o_orderdate >= '1994-01-01'
         AND o_orderdate < '1994-12-31'
         AND n_region_name = 'ASIA'
         AND o_cust_nation = n_nationkey
        GROUP BY
         n_name
ORDER BY revenue desc;
```

There are fewer joins involved, and we have eliminated the need for row shipping. This query will execute faster than the original one. Note that we have marginally increased the storage we need for the orders table and now may fit fewer order rows on the data pages of the underlying database. Large queries like those that require performing a sequential scan of the orders table may be now be slightly slower. Still, joins are costly, and we will have been able to dramatically reduce the time needed for other queries. It is up to the database designer to decide if this strategy should be employed or not, which will be influenced by the queries that are to be executed.

Denormalizing tables may also involve a little more work in your ETL process, but the extra effort will pay off with faster query times.

4.4 The Role of the Coordinator

The Coordinator does a bit more work than the other nodes in the system. It performs the optimization and query planning, as well as directs the execution of the query, synching up instructions for work going on at other nodes.

With this in mind, you may want to consider making one node a dedicated coordinator, particularly if your system has 16 or more nodes (it is not worth doing for an Standard Edition cluster). A coordinator that is dedicated would guide the queries on the other nodes but does not participate in them.

Coordinators can also house replicated tables. This allows for serving quick queries where a reporting app may, for example, get all of the possible values in a look-up table like region, while freeing up the other nodes to do the heavy lifting.

Again, the decision to go with such a configuration largely depends on the number of nodes you have. If just using a 4 node system,, it is not worth having a dedicated coordinator. You could however consider making it slightly more powerful than the other nodes (perhaps a slightly faster processor).

If you do choose a configuration with a dedicated coordinator, you merely need to take care to properly specify how the tables will be partitioned. For example assume you want node 1 to be the coordinator on a 16 node system. You would just specify the other nodes when creating tables.

```
CREATE TABLE tab1 (col1 int, …..)
PARTITIONING KEY col1
ON NODES (2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)
```

5 Redundancy, Backup and Recovery

The current version of ExtenDB is somewhat limited in this area, and functionality is currently being worked on for future support.

Some redundancy is achieved via a RAID configuration in case of hard disk failure, which is the most likely cause of a failure in a node. You can use software from the underlying database to create stand-by node database. For example, PostgreSQL offers a program called Slony for replication. Using Slony, a DBA could replicate all of node 1's databases to node 2, node 2's to node 3's and so on. In the event of a failure, the DBA would just need to change the xdb.config file to point to the stand-by database, with one physical node acting as two logical nodes.

Load balancing is achieved to a degree via parallelization. It is the execution of queries over large amounts of data that allows ExtenDB to speed up queries. Given low OLTP volumes, it may make more sense from a cost stand point to create the stand-by database nodes on existing nodes as described in the previous paragraph, rather than use entire stand-by clusters and load balance multiple clusters with multiple coordinators (though possible).

The current version relies on the backup and restore programs available for the underlying database. It is up to the DBA to manage that, optionally with the help of ExtenDB's utilities. This includes full backups and incremental or log file backups. PostgreSQL offers robust utilities in this area, including point in time recovery (since 8.0).