Open Sky Technologies

MPbase

M**assively **P**arallel Data**base

Where The Network Is The Database!

Complete Documentation Set

All Rights Reserved, © 1998, 1999
Next Paradigm Systems Inc.

An ongoing work in progress
March 2, 1999

Table Of Contents

Introduction 3
MPbase Is Just Like… 4
MPbase Fact Sheet 5
Real World Problem #1 "Sub Hunt" 7
Real World Problem #2 "Risk Check" 8
Real World Problem #3 "Profit Hunt" 9
Real World Problem #4 "Corp. Data" 10
Real World Problem #5 "E-Battlefield" 11
White Paper & Overview 12
How is this possible? 18
Architectural Options Diagram 19
Parallel Cluster Options Diagram 20
Task Vs. Resource Centric Processing 21
Virtual Metadata 22
Access Options 24
Content Addressable Memory 25
MPbase and SQL 26
MPbase And Application Complexity 28
Why MPbase Speeds Up The Whole System 29
Naturalization vs. Normalization 31
Boxes or Attractors 33
Data Types in MPbase 34
Removable Media 35
Year 2000 & MPbase 36
MPbase Architectural Limits 37
Product Plan & Deliverables by Phase 38
Building The Phone Book Demo 42
Phone Book Demo Statistics 43
Real World Numbers #1 "Data Transfer" 44
Real World Numbers #2 "Big DB" 45
Inventor's Notes on Paradigm Shock 46
Evaluating New Technology 48
Reasons To Use MPbase 51
MPbase, FAQ 52

Introduction

"Any sufficiently advanced technology is indistinguishable from magic."
Asimov

MPbase is based on such a *magic* technology. It is the first wholly new way to handle information since the creation of relational database theory. This new approach to information storage and retrieval ***avoids*** most of the classic database problems. What **MPbase** did not do, is solve the traditional database problems. What **MPbase** did do is avoid them.

This document is arranged in several sections. The order of the sections should help guide the reader through what the developer calls "paradigm shock". One section, Inventor's Notes on Paradigm Shock, is meant to be read out of order. This section should be read just after the reader thinks "NO #@*($% WAY," and just before this document is resigned to the round file.

The first section is titled MPbase Is Just Like… This section will attempt to convey some of the problems encountered in explaining something truly new. What do you do when there is no valid "A is just like B only…" How do you explain anything when there is no good common point of reference.

The following section is the wild and woolly MPbase Fact Sheet. It makes no apologies for stating the unbelievable. As a matter of fact it is subtitled "believe it or not." This section is guaranteed to extract a derogatory expletive from any knowledgeable reader. But as the section says, everything stated there has seen real world proof.

The next several sections are titled Real World Problem #… These are examples of how **MPbase**'s unique characteristics can be applied to solve some of the more difficult database problems. Each one will, again, most likely evoke the response, "if that were only possible, it would be a good solution."

This leads to the next section, White Paper & Overview. This is where the characteristics of **MPbase** are revealed in a little more detail. This section provides the reader with a general overview and a feel for **MPbase**'s unique capabilities. This is followed by several one to two page sections each dealing with a specific capability or issue in more detail.

Next is the section titled Product Plan & Deliverables by Phase. This section explains where **MPbase** is now and where it is going. This section also contains a brief history of how **MPbase** came about.

After the plan are the sections discussing Open Sky's demos, describing how they were created and some performance statistics. This is followed in turn by two sections to help deal with "paradigm shock." Finally there is a list of reasons to use **MPbase** and a brief FAQ section.

MPbase Is Just Like…

What's wrong with this statement?

MPbase is just not quite like anything you may have come across before. It has many characteristics that are truly unique. It also has many characteristics that it shares with current data handling environments. Most people will look at a single **MPbase** characteristic and say "Oh that means **MPbase** is just like ________." From this myopic, "single characteristic" point of view they will normally be correct.

However, it is important to consider that at its core **MPbase** is a unique entity. It is neither fair nor safe to make assumptions based on single matching characteristics. Normally certain characteristics are reliable indicators of others. In other cases certain characteristics are mutually exclusive. Most of these conditions do not hold true in the **MPbase** environment.

The following paragraphs describe many of the major characteristics of **MPbase**. At first read they may seem to be a set of impossible combinations. It is possible! This is due to the unique internal architecture of **MPbase**. The only possible way to describe **MPbase** is with a large and inaccurate set of "sort-of-likes." Maybe someday there will be some "just-likes." But for now "sort-of-likes" will have to do.

MPbase internally is sort of an …

> Update-efficient, highly compressed, content-addressable, multidimensional, data-intelligent, relational, hierarchical, object-oriented, platform-independent, self-organizing, massively parallel, self-tuning, fault-tolerant, network-centric, hypercube.

Also it is sort of …

> Index-free while at the same time being just one big index. The fastest possible structure for access and/or update. Best used for transaction-based processing and/or best for batch processing.

MPbase is precisely none-of-the-above. But it is sort of all-of-the-above. In addition, the more data and tasks you give **MPbase** the better it works. The better the compression works. The better the update processing works. The better the batch processing works. The better the transaction processing works. The farther **MPbase** gets behind, the faster it will catch up.

The combination of characteristics in **MPbase** only make sense from the inside looking out. From the outside looking in there seems to be no possible way that this combination of characteristics could possibly come from one system. Internally **MPbase** has changed all the rules. Therefore all of the normal assumptions need to be revisited. Most of the limits to current information processing are self-imposed. It is time to lose those limitations and start using the full power of the underlying hardware.

MPbase Fact Sheet

Believe it or not!

Normally technical pieces are decidedly understated, and hence, defensible. In the case of **MPbase,** even this understatement can look like a set of outrageous marketing claims. Because of this, I will now lay out for the reader a quick set of the features and functions without any attempt to understate them into a more believable form.

As wild as the following sounds, it is all provable. Everything stated here is a FACT. Everything stated here has seen at least one working example. So, here is **Open Sky Technologies'** presentation of **MPbase's** fact sheet, **believe it or not!**

Initial load & index creation

MPbase has no separate mass-load function. All rows are loaded into **MPbase** using the update function. The update function is the most efficient possible load method. So no mass-load function is required. Because mass-data-loads use the update function, index creation is an automatic and integral part of the process. Because of the nature of the load process, it is NOT physically possible to see an inconsistent image of the database.

Backup/Restore

Because within **MPbase** it is not physically possible to see an inconstant image of the data, backup/restore processing can safely be done in segments at any time. Using the roll-forward and roll-backward logs, such a piecemeal backup can be restored as a fixed-point snapshot of any time between the starting and ending point of the backup.

Reorganizational Unload/Reload

When **MPbase** is running on a cluster with mirroring enabled, reorganizational unload/reloads can be done without taking the database down. User queries and even updates can continue throughout this process. The new copy of the database can be verified and then can be accepted or rejected without the users ever knowing of its existence. Once the new copy is accepted, **MPbase** can switch to it without any disruption to the user queries.

Compression + Encryption = Speed

The manner in which **MPbase** stores the data is highly compressed and encrypted. This is done to increase the data access speed. A byproduct of this compression is the saving of disk space. Typical compression rates for **MPbase** are on the order of 90% or better for non-image ASCII data. Image data rates are only about 50% for the lossless compression of single-frame scanned images.

Loosely coupled parallel clustering

With **MPbase** the database image is decoupled from the hardware. This allows an **MPbase** cluster to be fault tolerant at the machine level. This means an **MPbase** cluster has NO single

point of failure. Any machine in the cluster may be shutdown at any time without effecting the function of queries or updates. This includes inflight transactions as well as any new transactions.

Big Binary Blobs

In **MPbase,** the concept of big binary blobs has no meaning. **MPbase** can be set up to handle any type of data blocks in an intelligent manner. Queries of image data can be made at the pixel level. This allows questions to be asked of the internal content of such an image. Questions such as, Are there any discontinuities at a level below the screen resolution? In other words, On this x-ray are there any breaks not visible on the screen?

Fully Predictable Scaling

With **MPbase,** the black magic of scaling goes away. If you have **MPbase** in production and you suddenly find you need 100 times the capacity, no big deal. You can place one hardware order with 100% confidence in the final performance numbers you will get. If there is a surprise it will be on the high side. You will get more than you thought you would.

Isolate Interesting Data

One of the aspects of "Multidimensional Data-Intelligent Run Length Encoding" (MDIRLE) is its inherent ability to isolate "interesting data." As MDIRLE uses a content addressable memory (CAM) schema, data that does not conform is isolated. This data is inevitably one of two types, data in error, or interesting data. In either case, it is data you need to know about.

Quick System Development

What is IT? It is ***Information*** Technology. What is a database? A technology that handles ***information***. If a database was powerful enough it would **be** the IT department. **MPbase** is everything you need except the user front-end. A new system equals a new front-end. Everything else can be handled in **MPbase**. As the information-handling tasks can now reside where they belong, the application becomes a very simple front-end GUI.

The Theory Of Relativity

Everything is relative to the viewpoint of the observer. The **MPbase** view can support any possible view of the information in an **MPbase**. This means a system requiring a relational view can "see" a relational **MPbase**. A system needing an indexed file system can "see" an indexed file system **MPbase**. A system needing a multidimensional cube can "see" a multidimensional cube **MPbase**. A system needing a hierarchical database can "see" a hierarchical **MPbase**. All of the systems can be using the same **MPbase** at the same time. The view has nothing to do with the physical storage formats used in **MPbase**. It is all relative.

Real World Problem #1 "Sub Hunt"

Where did the submarine go?

The problem is to be able to track a single sub in the ocean of information coming from the undersea passive sonar network. This network produces sample records at a rate of 10,000 every 2 seconds, consisting of 64 16-bit numbers (FFT), a location, and a time/date stamp. This large volume of data must somehow be stored in a database. The database must then be able to find all of the records within a time range that have soft matches to a target FFT. This output set must then be displayed as a track on a map. This answers the question. Where did this sub go?

This problem can be solved by using **MPbase**. Although this is a complex data problem, it a simple information problem. **MPbase**'s combination of internal characteristics make easy work of this task.

First, the multidimensional hypercube nature of **MPbase** allows the relevant data to be logically located near each other in all of the 66 dimensions (64 numbers, location and time). This entire database would need to be stored in only one **MPbase** table.

Second, the content-addressable memory (CAM) nature of **MPbase** will cause all of the records with similar FFTs to be stored near each other in the hypercube.

Third, **MPbase**'s ability to do a soft match makes locating the needed FFTs a simple task. This can be further enhanced with the addition of custom match or analysis routines. These routines can be used to augment and/or replace **MPbase**'s own internal routines.

Fourth, the massively parallel architecture allows for the sheer volume of updates that such a database will need to accept. **MPbase** has no special "data load" function. None is needed. All data placed in **MPbase** goes through the update cycle with no performance penalty. As a mater of fact, the more updates **MPbase** has to process the more efficiently it does so. The farther behind it gets, the more quickly it will catch up.

Fifth, data in **MPbase** is internally compressed to a very high level. This, coupled with the parallel nature, allows all of the needed performance to do this task in real-time.

Sixth, the relational nature and formatting functionality in the output view allows **MPbase** to provide the result set in any needed format. This could directly feed a GIS system or even a CAD program to graphically display the sub's track.

Seventh, the fault-tolerant nature of **MPbase** provides the kind of reliability such a system must have. If enough hardware remains to run a query, **MPbase** will continue to function

In short, with **MPbase** this nasty real world problem is no problem.

Real World Problem #2 "Risk Check"

Determine which parts are at risk.

This problem is aircraft-part tracking. It is to determine parts or assemblies that may be at risk after a specific part or assembly fails. As the parts move through the factory each develops a history. This includes many batch numbers one for each operation performed in manufacture. Any part not meeting the specifications for an operation will move to a new batch for rework. This creates a large and tangled web of interactions.

The problem query is to find all parts that have ever shared any single batch with the target part. The database must then allow questions to be asked about all parts sharing that batch, including where they are now and how many other shared manufacturing operations or components do they have.

This query must also allow qualifications to limit the search to likely candidates. This query must range from as narrow as parts that had threads cut with a particular die, to as wide as any part or assembly containing metal from a particular foundry.

When you look at all of the parts and assemblies in all of the possible aircraft, this is a monumental task. Such tasks are perfect for **MPbase**. The combination of internal characteristics found in **MPbase** make this a doable, if not an easy task.

First, the multidimensional hypercube nature of **MPbase** allows the relevant data to be logically located near each other. In this case only two linked cubes would be needed; one for the part or assembly and the other for the operation and batch.

Second, the content-addressable memory (CAM) nature of **MPbase** will cause all of the part records with similar histories to be stored together in the hypercube. Also, all of the similar operations will be stored together. This allows a relatively easy traversal of this complicated net.

Third, the massively parallel architecture allows for the sheer volume of data that such a database will need to store. This is helped significantly by the internal compression of the data. These two characteristics provide the needed performance.

Fourth, the search could be expanded to include all parts or assemblies that share similar operations or tools. This would allow the search of all parts that were ever touched by a specific die or tool, even if they did not share any common batch.

The natural information-based clustering found in **MPbase** makes short work of this type of very tricky data problem. It does so by turning it into a simple information-based problem.

In short, with **MPbase** this nasty real world problem is no problem.

Real World Problem #3 “Profit Hunt”

Where is my profit?

The business is an online reference service. The problem is to be able to track the cost and income streams associated with billions of documents across millions of users. The source of this information is about 15 million transaction records per day. In addition to user and document numbers, many different subtotals are needed by such things as, sales rep, state, database, library, market, tier, file, branch, country, firm, affiliate and service.

The current solution is based on a whole lot of batch processing. The first step is to filter and summarize the raw input on a nightly basis (information loss #1). The next step is to produce/update a set of databases IDMS, DB2, and others (information loss #2). Once these databases are ready a subset of information can be extracted from each (information loss #3). These extracts are processed monthly to produce a multidimensional database (information loss #4). Finally, questions can be asked. But only of information that actually makes it into this multidimensional database and only on the monthly snapshot.

In contrast, the **MPbase** solution contains four parts:

1: Load the raw data directly into an **MPbase**. This is possible because the nature of **MPbase** allows it to accept high volumes of relatively unfiltered data. The process of loading tends to highlight any errors in the information. This allows for the isolation of errors without the normal information loss.

2: Set up **MPbase** views to feed and/or support the IDMS, DB2 and other database systems. **MPbase** can appear to be almost any type of database. This allows two options in dealing with the currently running systems. The first is to produce an extract in the optimum format and order to load into the current database systems. The second is to create an **MPbase** view that allows the application to think it is still talking to the old database.

3: Load any plans and forecasts into the same **MPbase**. **MPbase** has the unique ability to deal with different levels of information in the same database at the same time. This allows direct comparison of raw detail with planned totals. It is only necessary to load the lowest level planning numbers as **MPbase** can produce the higher level totals internally. Currently where there are separate weekly and monthly numbers, daily numbers, if available, could provide both.

4: Use the **MPbase** directly to answer the same multidimensional questions in much more detail. Questionable records may be included or excluded in the viewed totals. Questions would be answered with real time data with no delays waiting for batch work to complete. Any total at any time could be drilled down to its lowest level components. This puts an end to the long and tedious process normally required to “prove” a number produced from a traditional multidimensional database system.

In short, faster access to better information with **MPbase** this nasty real world problem is no problem.

Real World Problem #4 "Corp. Data"

Data Warehousing & Data Mining

Imagine for a moment a computing wonderland. Where the least expensive solution is the fastest and the most reliable. A land where any level of performance is possible. Where doubling your capacity means doubling your cost, period. A land where development time is measured in hours and days, instead of months and years. Where first prototypes can safely be used in full scale production. A land where change is painless and controlled. Where complete code testing is not only possible but practical.

How is this possible? By avoiding many of the common DBMS problems.

- No need for metadata. It contains less information than the original.
- No data normalization. Normalized data is harder to mine.
- No conventional DBMS. There is no operating system on top of your operating system.
- Scale predictably, with a better than linear cost/performance curve.
- Eliminate barriers to growth with ***no*** architectural upper limits.
- Save money, cost savings between 10 to 1 and 100 to 1 are possible.

Other than avoiding problems, what will **MPbase** add to the picture?

- Increase reliability by using a paradigm that allows inexpensive parallel redundant systems. 100% uptime can be both practical and cost-effective.
- The speed to strip mine your raw data; conventional data mining is like taking core samples and working with averages.
- Demystify your data by storing it in a more natural format, one that highlights errors and discrepancies for easy identification and correction.
- Gain controllability with a paradigm that allows exhaustive testing and parallels.

How about access?

- Easy powerful interfaces to Visual Basic, HTML, PERL, C, UNIX command line. Almost any system can directly access a **MPbase**.
- Put the data on your intra-net to allow any system you authorize access to the data.

Real World Problem #5 "E-Battlefield"

Any database out in the real world

The problem is information. How to acquire, validate, store, analyze and distribute information within a distributed network. Lots of little pockets of information held by different people with different needs and goals. All points outside the network and random points inside the network must be assumed hostile. Add this to nodes in the network playing a constant game of hide-and-seek, and you have a problem no conventional DBMS can solve. Whether this is a military or corporate battle field the problems are the same.

MPbase has several unique characteristics that make it ideal for this very challenging environment. In a traditional DBMS, the functions listed above (acquire, etc.) are largely separate. Each part of the solution using different methods and tools. With **MPbase** all of the information handling requires only one tool, **MPbase**. There is no one-to-one correspondence between features and the required functions. Instead the problems are solved by the way **MPbase** deals with the information itself.

First, the content addressable memory schema allows for distributed acquisition, validation and analysis. As an example, each field node could contain the data for X hundred yards in any direction. The holder of this node can now add to, or correct, information about this area in the networked knowledge base. Many such overlapping circles of information can exist in a single **MPbase** without creating any problems.

Second, each **MPbase** is already made up of many little TCP/IP connected databases. In the phone book demo alone there are over 80,000. Small sets of these databases can live in network-connected laptops, palm tops, or backpacks. These little DBs can be moved around "at will" without confusing **MPbase**.

Third, encryption can be used at the lowest level of the DB. This after a compression that is a good encryption by itself. Because the search engine works on the compressed buffer directly, "plain text" need never exist outside of the final result set.

Fourth, an **MPbase** can be configured as a fault-tolerant hierarchy of information. This allows the best available information to be used in all decisions. The best source is a node at the site. The next best, is the node real-time synched at last contact. Next, a local interconnect. Last of all, a system wide interconnect.

With **MPbase** this real world environment is no problem. The natural information-based clustering found in **MPbase** makes short work of this information-based problem. Every **MPbase** already calls this type of complex networked environment home.

In short, with **MPbase** this nasty real world environment and problem are no problem.

White Paper & Overview

Open Sky Technologies has developed a method by which databases can be created using the native operating system as the core of a database management system (DBMS). Such a database architecture when using a massively parallel cluster of I/O processors, has ***no hard limits*** to speed, size, maximum users, reliability, or recovery options. No architectural limits to when or how much capacity and/or performance can be added to an I/O cluster. It is now possible to start considering your corporate information based on business need rather than on computing or traditional database constraints.

CUSTOMIZATION

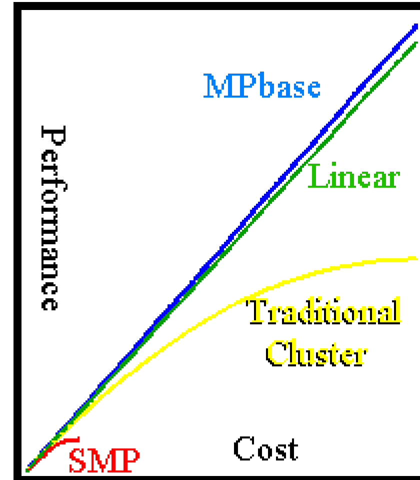
In traditional DBMS's, a single set of difficult tradeoffs is the inevitable result of any database design. This means that design considerations have been driven by the *database's* limitations. The Open Sky Technologies approach puts the focus on the system goals, not on the limits. Each new component builds on, or in parallel with, the rest of the system. Each **MPbase** is created to grow and change without the normal pain factor. The method used to access the data is neither fixed nor limited to a single choice. The **MPbases** developed to date have used three or more parallel access methods to a single copy of the same data at the same time. Each individual access method can then be tuned and optimized for the applications it is supporting.

DBMS Design Questionnaires

Traditional:	MPbase:
Select one point in each range	Indicate importance of each **All five's OK**
Online Batch 0 1 2 3 4 5	Online (0,5) ______ Batch (0,5) ______
Recovery Through-put 0 1 2 3 4 5	Recovery (0,5) ______ Through-put (0,5) ______
• • •	• • •

PERFORMANCE SCALABILITY

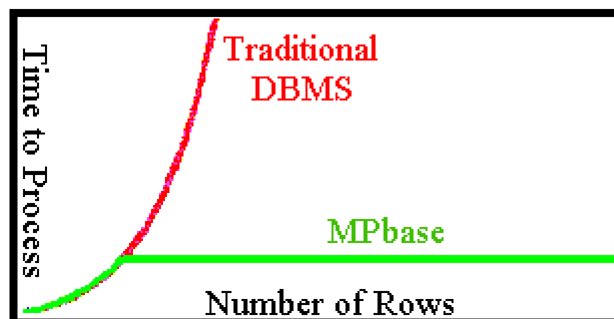
With most clustered architectures, linear scaling is a much sought after, yet never achieved, computational Holy Grail. **MPbase** generally exhibits ***better than linear*** scaling. The more I/O processors an **MPbase** cluster contains, the faster each processor will run. This is only possible with an architecture driven from the bottom up (pull) not the top down. As the number of I/O processors continues to increase, the scaling may eventually fall back to *linear*, but never less.



Performance of an **MPbase** I/O cluster is primarily determined by the amount of disk space given to each I/O processor node (disk/processor ratio). This factor is determined independently of data capacity and access capacity, and can be changed independently as well. This means that a performance level can be set that will not degrade as more capacity is added to the **MPbase**. This ratio determines the performance for the worst case query. All other cases will show improvement with size. With **MPbase** the bigger it is, the faster it runs!

DATA SCALABILITY

There is no performance penalty for adding hardware to an **MPbase** I/O cluster. There is no architectural upper limit to the size of a table in an **MPbase**. After the initial disk-to-processor ratio is set, the number of disk/processor sets (nodes) needed is then determined by the amount of data to be stored. This means that to add more data, add more nodes. A single query will never be slower after adding more nodes and will most often run faster.



The two-part Achilles heel of most conventional large databases is ***fragmentation*** and the ***reorganization*** required to correct it (or, for that matter any process requiring a full sequential scan). With **MPbase**, once the disk/processor ratio is determined, the time needed to perform any full database process will be no more than the time needed at a single node. This time will ***NOT*** increase as the database grows. A table of 100 billion rows will reorganize as fast as the number of rows found on a single node. Once again, as **MPbase** grows, the worst-case-processing times will not increase. All other processing cases just keep getting faster with size.

ACCESS SCALABILITY

The architecture of **MPbase** allows a true many-to-many network between the nodes holding the data and the nodes providing user access. This means **MPbase** has no single choke point. This lack of a single access/choke point allows the I/O cluster to be scaled to ANY level of parallel access. This also allows parallel access to scale independently of data scaling and performance scaling.

The architecture allows a single database image with no single choke point. This means creating a database with ***NO LIMITS***. A database which will start at any requested level of performance. A database to which you can predictably add any amount of storage, parallel access and/or performance. A database which can start small and gracefully grow to ANY size providing ANY desired performance level with ANY degree of parallel access.

RELIABILITY

MPbase can be configured to be fault-tolerant at the node level. With a massively parallel platform, fault tolerance is not an option. **MPbase** can be implemented with no single failure point. The loss of ANY one user or data node in the **MPbase** I/O cluster need not stop it from functioning nor cause any data to become inaccessible.

The architecture is such that an **MPbase** need never go down. Backups, restores and even re-organizational unload/reloads, do not require queries or transaction processing to be stopped. Depending on requirements and **MPbase's** I/O cluster design, even updates can continue during such processing.

DATA COMPRESSION

All data and data types stored in **MPbase** are kept internally in a highly compressed common format. The compression method, which is lossless in nature and unique to **MPbase**, is called *"Multidimensional Data-Intelligent Run Length Encoding."* Think of this as computer-readable shorthand. This allows ***even faster*** access to the compressed data than would be possible for the uncompressed data. Typical compression rates can be 60% to 99% over a flat ASCII (text) file. The resulting physical format is directly usable on *any hardware* capable of running **MPbase**. The compression format by itself provides a good level of both security and data integrity. If required, the low-level access routines can include varying additional levels of encryption parallelized across multiple nodes. The function handling the compression also provides access or update monitoring and/or control through a single low-level routine. Access to the database can then be as open or as controlled as required.

STORING DATA AS INFORMATION

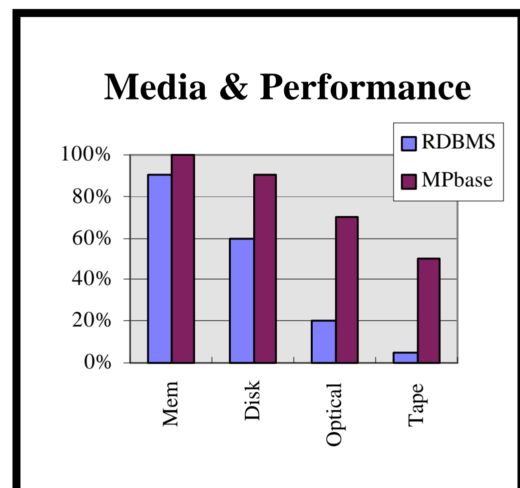
In **MPbase**, the data is transformed inside the low level objects into information. This can best be described as a formless format. From this state, the data can be recreated in any required output format. There is no performance penalty for this activity. As a matter of fact, this reformatting is used to increase performance while at the same time significantly decreasing the physical storage requirements.

This data storage technique allows the direct linking of any tables containing the same "information." Internally, the external format is irrelevant. The key values are meaningless until converted by each table's access code. This puts all of the information stored in **MPbase** into the same information space. The only time the external data format is used is in the final result set to be sent to the user, without regard to EBCDIC vs. ASCII vs. proprietary formats. On the inbound side, the external format is lost just as quickly. It is converted to an internal transfer format before being passed to the data nodes for storage in the formless common format.

STORAGE MEDIA PERFORMANCE

The storage media model changes significantly with the implementation of **MPbase**. It is possible to view optical or tape media in a much improved light as they relate closer to the performance of magnetic disk and memory.

In comparison with conventional databases, memory is faster, disk is as fast as memory was, and optical is faster than disk was.



DISTRIBUTED MODEL

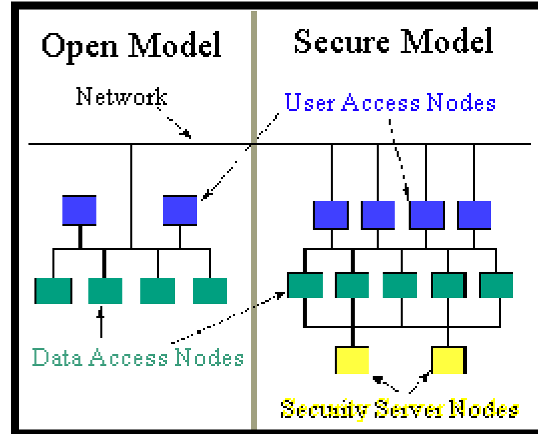
The loosely coupled architecture lends itself to a unique set of implementations. For instance, there is no requirement that the nodes making up an I/O cluster be in the same room or even at the same site. This same freedom exists in relation to second or even third copies of the same data when running mirrored. Think of the possible benefits to having a single consistent database image based on hardware placed all over the world. Each branch could maintain its own portion of the corporate data, while at the same time corporate headquarters would have a company-wide view. The possible configurations are as endless as real world business requirements dictate.

For extreme reliability, **MPbase** can be run with distributed mirrored copies at multiple sites with each site kept in update synchronization in real time. The queries can be distributed to the site currently running with the least load. This distribution can occur at the sub-query level allowing all sites to work on parts of a single query as the load permits. In this configuration the failure of any one node or even one entire site need not affect the ability to continue to run queries.

SECURITY

Security in a networked environment is a major concern. **MPbase** has two operational models to address this. The open model which is used with non-secure data and/or on a protected internal network. Or, at the other end of the spectrum, the secure model can be configured to meet any possible requirements.

In the open model, the access routines can be stored and backed up with the data requiring no keys and operating from the command line. In this mode, the compression functions as just that: compression. The only side benefit is data integrity. Think of this model in the same light as a self-extracting archive. If it is corrupt, it will tell you. If it is not corrupt, it is usable on its own.



At the high-security end, the compression can include multiple layers of encryption. The access routines can be kept on a single high-security server within the cluster. Access can require external keys which can be passed into the cluster encrypted or found only on a second high-security server within the cluster. The cluster can be configured so that the security servers are not accessible or even visible outside of the cluster. The way the database functions, no user logins need ever be allowed directly on the cluster. All access can be through secure Remote Procedure Calls (RPCs). In this mode, the user access nodes function as firewalls (secure access points). Backups can be done in sections, such that each section is useless without the others. The backup sections can then be stored at different sites. **MPbase** can search, total and analyze the data without first producing the plain text.

HARDWARE COST

And now, really good news! **MPbase** loves small inexpensive workstation class machines. Because of the fault-tolerance at the node level, there is no need for special high-reliability fault-tolerant equipment. Also, because there is no penalty for adding nodes, there is no need for the premium-priced, high-end SMP platforms. The main criterion for an efficient hardware selection is dollars per unit of useful work.

When redundancy is required, the combination of the compression and the ability to use less expensive hardware really adds up. A configuration using two copies of the data will typically have an aggregate compression (including the space used by both copies) of 60% to 80% or better. Based on this type of compression multiple mirrored copies become a very cost effective option.

ARCHITECTURE

The "magic" that makes all of this possible is a massively parallel, object-oriented architecture that can best be described as inside-out. The database exists, not "inside" a black box, but "outside" in the operating system's environment. The **MPbase** architecture is much like the operating system (OS) itself. It is made up of lots of little independent pieces. Each of these pieces communicates to the others through the file system interface. The result is a system that can take full advantage of all of the OS features and functions.

What Open Sky Technologies has developed is a method by which such a database system, spread out across several machines can act as if it were a single black box. At the same time allowing the system to run as fast as if it were an embedded or stand alone system. When used with *"Multidimensional Data-Intelligent Run Length Encoding,"* the resulting performance is nearly unbelievable.

ACCESSABLILITY

Due to the "inside-out" architecture this database is directly accessible by any program or system that can use the OS file system interface. It has been accessed by using: Structured Query Language (SQL), World Wide Web (HTML), shell scripts, command line and custom programs of any language. Cross platform access is accomplished using HTML, RPCs or sockets. All of these are available on virtually any computing platform.

EXAMPLE

And now, some details for those into numbers. One **MPbase,** built for less than $500,000.00 on thirty workstation nodes, contains over 50 billion 84-byte rows. It is designed to grow to over 150 billion. This database runs 24 hours a day, seven days a week, with no downtime. The access rate is over 90 thousand rows, selected, sorted, and returned per second. This rate involves a key hit rate of only 25%. So the database is handling 360 thousand keys per second. This extract rate is maintained in parallel with a 150-row-per-second insert rate, and a 50-row-per-second update rate.

The internal processing rate, available to "next generation" applications, is over 5 million rows per second, or 18 billion rows per hour. The compression achieved in this database is 95% for a single copy. Counting both the primary and mirror copies the compression is still 90%. This database system lacks any single failure point.

How is this possible?

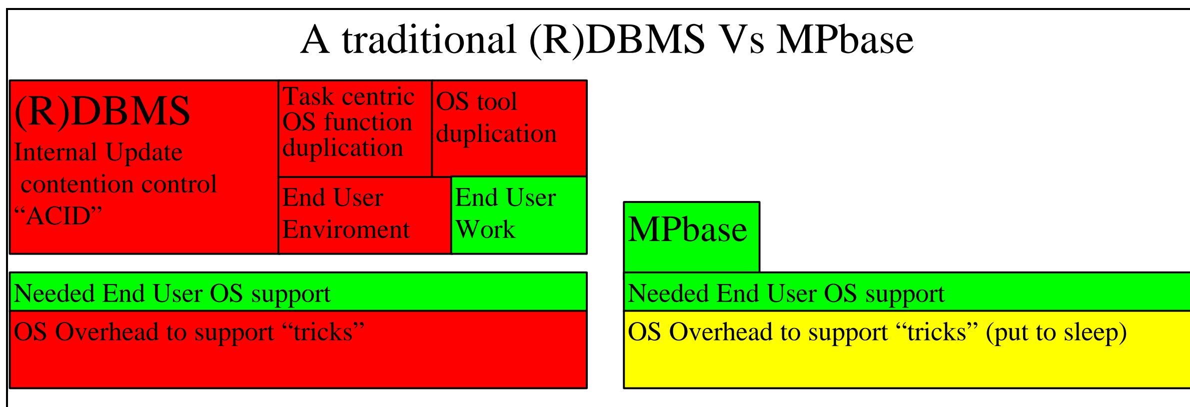
No car goes 1500 MPH

MPbase is capable of performance that should not be possible given the hardware. How is this possible? The answer is simplicity. A simplicity that is only possible with a resource centric architecture and content addressable memory (CAM) schema. Computer systems have over the years developed into an unbelievable level of complexity. Systems have layers upon layers of code. Each layer's goal is to hide the complexity of the layers under it. This hiding makes things *APPEAR* to be simpler, when in fact each layer actually makes things more complex.

MPbase is only possible due to an understanding of all parts of the DBMS problem. They include, the data manipulation itself, operating system architecture, and the underlying hardware architecture. Not long ago such a layer free system would be highly closed and proprietary. Today, however, both the OS, and the hardware have been standardized to the point that this is not only possible but practical.

Most parts of an (R)DBMS are only needed to internally support the layers. Very few of the parts are actually doing end user useful work. It is amazing how little database functionality is really required to directly support the end user. Working with the OS and using a CAM schema are the keys to removing (R)DBMS layers.

Much the same can be said for the OS. Most of the OS is there to support the rest of the OS, not the real end user useful work. Most of the internal OS functionality is there to support "tricks." Each trick is intended to speedup the system. However, each trick adds to the system complexity and overhead. The solution is resource centric processing.

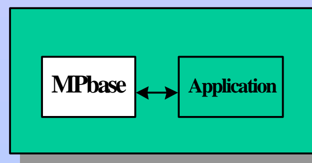


MPbase is capable of doing the same end user useful work with far less system effort. On any given hardware platform there is only one way to truly speed a system up. DO LESS WORK. The way to reduce the workload is to understand what tasks are ***not*** essential to end user functionality and minimize their impact on the system.

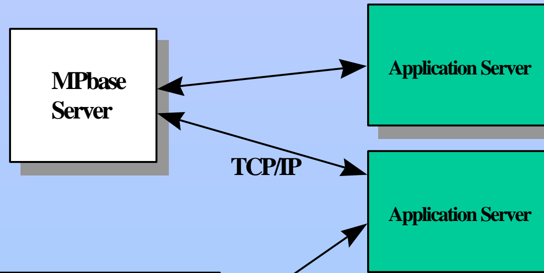
Architectural Options Diagram

Open Sky Technologies
MPbase Architectural Options

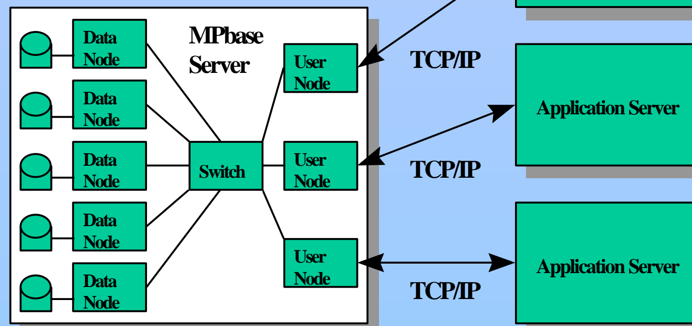
Single "box"



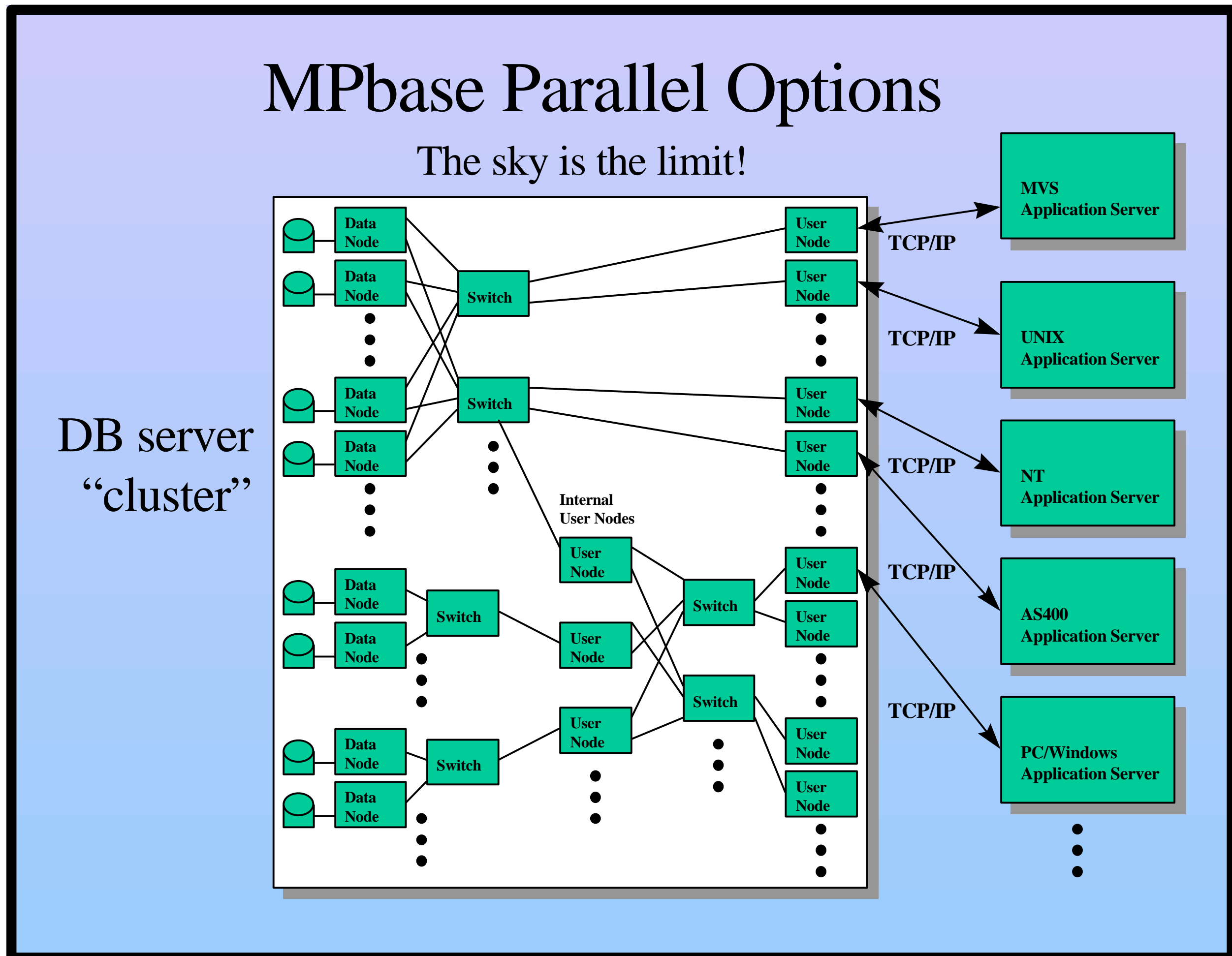
DB server "box"



DB server
"cluster"



Parallel Cluster Options Diagram



Task Vs. Resource Centric Processing

The two computing mountains

Contributing to the performance of **MPbase** is a fundamental change in the way the computer hardware is used. This change is to a resource-centric paradigm. This leads to a whole new set of performance constraints. These constraints are far less restrictive than those from the current task-centric model. This allows far greater performance than would otherwise be possible.

Almost without exception, current computer-based systems run in a task-centric paradigm. Under this paradigm the task to be done is the central focus. The system resources: disks, memory, I/O connections, etc. are queued up and shared between the tasks. Queuing theory describes, in agonizing detail, why this paradigm can run only just so fast.

As long as systems remain task-centric, the theoretical upper limit to performance will be described by queuing theory. This artificial upper limit to performance is nowhere near the true limits of the underlying hardware. Prior to **MPbase** and this conceptual breakthrough, performance like this was only possible through stand-alone or single-user systems. Although very fast, such systems are not very practical.

Resource-centric systems behave in a very different manner. In this paradigm the task is fragmented and placed in resource queues. Each resource is owned by one system task (daemon). All the work requiring use of any one resource is done by the one daemon owning it. The results are fed back to a daemon owning the "task," this last daemon combines the products of the resource-owning daemons to produce or validate the completed "task."

To some extent all systems do contain some of both paradigms. However, there is a line beyond which the resource-centric model "short circuits" queuing theory. This is where the full power of the underlying hardware is available for end-user useful work. This is the realm of one-to-one decoupled queues. Some would claim that this environment is as odd and hard to use as stand-alone mode. However, **MPbase** allows this operational model to be applied to real world information problems.

A good example comes from several years ago. Mirroring was being added to a system. The sample run took one hour before the change. The change was first added in a task-centric manner. This took the run time to two hours. That change was then removed and reapplied in a resource-centric manner. This took the run time to one hour and five minutes.

This example was not done just to validate the resource-centric model. It came about due to a combination of disbelief and developer brain fade. A little bit of task-centric behavior cannot be all that bad? WRONG! Once the new rules are understood, this is not a hard paradigm to follow. When you make a mistake it tends to be easy to find and almost always falls in the "that was dumb" category. Not counting the test runs, all of the changes above took less than one hour to complete.

Virtual Metadata

The keys to the information world

First, let's establish a working analogy for metadata. Metadata can be viewed as the information on a library's card catalog. This represents a very small but common subset of the information contained in the complete library. The catalog's information subset can be very useful both on its own, and as an index to the main body of information. The task of creating or changing such a catalog is a large and costly one. This cost severely limits the usefulness of a card catalog or any set of metadata.

Now, what if you could use a "magic" card catalog? One where all you needed to do was define the card content and start your search based on your own definition of what should be on the cards. This is virtual metadata. A database view containing summary information about the main body of the database. A view which, like metadata, can be used on its own or as an index to the main data store. A view that is painless to build or change.

When you start looking at computerized metadata the analogy starts to break down a little. In some systems the metadata storage is several times larger than the original data. This is where you really need to start asking, "Is this the best way to work with this data?" In the case of more metadata than data, there are normally millions of subtotals generated most of which will never be used. They are there to support queries that may or may not ever be run. The reason all possible subtotals are generated is that, at the time of creation, there is no way to know which subtotals will be needed.

So, why use metadata at all? If the database is fast enough, as in the case of **MPbase**, why not just work with all of the data? Very few database applications can handle the totality of data in a data warehouse. Even if the database can produce all of the data in a timely manner, it would just overwhelm the application (like filling a teacup from a fire hose). With **MPbase,** meta-data is still required to reduce the data volume to the application. **MPbase** can produce this needed metadata on the fly so that it requires no additional storage space.

This virtual metadata is defined as a database "view". A database view is nothing more than a way of telling the database what you wish to see. A view can be thought of as a virtual database containing a subset of the complete database. In addition, a view may contain summary information from the database. In short, the view is used to create the "magic" card catalog mentioned above.

As an example, take a company with a 1-terabyte database and 3.5-terabytes of metadata. **MPbase** could reduce the 1-terabyte to 300-gigabytes and eliminate the need to store the other 3.5-terabytes. The total savings in this case would be 4.2-terabytes.

This virtual metadata from **MPbase** is the perfect way to use a massively parallel database with a "normal" application. It allows the database to do what it does best and leaves to the application the tasks that it does best.

One key aspect to using "virtual metadata" is in the way you handle "big binary blobs" (BBB). This is a traditional database's way of handling data it does not understand: things like image data. **MPbase** can work with the information inside the BBB when creating the "virtual meta-data." This allows queries directly into the content of the BBB. In the traditional database environment, this would involve a separate application reading the BBB's and producing fixed metadata.

Access Options

Welcome to a truly open system

MPbase is a truly open database environment. It is also very network-centric using TCP/IP as its primary method of communication to and from the physical data. **MPbase** gets its performance from its parallel nature and its efficiencies and can afford to be network-attached and still achieve industry-leading performance.

If the application can talk to a UNIX socket, Telnet, HTML, XML, or in any way communicate with the UNIX command line, **MPbase** can do the rest. Part of the definition of an **MPbase** "view" is any needed protocol and/or formatting of both the request, and the result-set.

This allows **MPbase** to be used in environments where the calling program's request format and result-set format cannot be changed. This also allows different environments with different communication requirements to communicate to the same **MPbase** at the same time.

Finally, a single database usable by all of the computer environments in your shop!

Here is a real world example. A single database was accessed by MVS (TSO/Batch), HPUX (HP's UNIX), Solaris (SUN's UNIX), AIX (IBM's UNIX), HTML (WWW / Internet), Windows 3.11/95 (BSD Socket). In addition each "view" supported multiple versions of the application code including changes to both request, and result-set formats. ***All from the same database at the same time!***

In the future **MPbase** may even add direct database access through the NFS interface. This would allow the DBA to set up "flat file" database views mountable through NFS. This would completely eliminate the need for the applications to even be aware of the database. These views could be "canned" queries set up to feed current systems' real time information.

Content Addressable Memory

No data questions, just answers

MPbase's internal structure is in part based on the concept of content-addressable memory (CAM). In this information-handling model, each possible piece of information has one and only one possible storage location. The data is its own key. It is important to differentiate CAM from a hash key or traditional index.

With conventional indexing schemes the data content is used with a hash or index to produce the address location of the data. The address has no real or direct relationship with the information contained in the data. With CAM, the data describes its own storage location. This also means all like data will always be found close together in the physical data structure. There is a direct relationship between the information in the data and its location in the physical data store.

What can this mean for a database? First, any piece of the data can be used to narrow the search area. Second, all similar data will be found in close logical proximity. This means that the data logically closest to any row will be, by definition, the most similar to it. This makes analysis of the data a much simpler task. It also speeds the updating process.

With CAM, the final location of any row is predetermined. Normally the determination of the physical placement is a time-consuming task. This is why most databases use a separate process for loading the database. The load process can make bulk decisions, and so run much faster than the normal update cycle. With a CAM-based database the data will, to a large extent, "sort itself out" during the load process.

With an appropriate database architecture using the CAM model, there is no need for a special load process. The update cycle is just as efficient as a batch load process would be. Therefore this type of database is perfect for mass streams of data that will need to be analyzed. Not only will the update run like a batch load, but a good portion of the processing needed to analyze the data is already done.

Why isn't CAM the standard model for database architecture? The physical storage models currently in use require the data to have certain characteristics. The tight coupling between logical and physical location in current DBMSs requires data to be uniformly spread across the physical media. To allow the data to "bunch up" is highly undesirable because it creates choke points.

In order to use the CAM model, it was necessary to use a physical storage schema that could benefit from this bunching up. **MPbase** does just that.

MPbase and SQL

Everything needed and no more

This paper endeavors to provide the information needed to form your own opinion about a fairly complex issue; **MPbase**'s support of Structured Query Language (SQL).

MPbase has included sufficient SQL support to allow systems that need or want to communicate in SQL to do so. **MPbase** provides efficient support of several inefficient interfaces. The **MPbase** view, which is an extremely enhanced version of the traditional database view, allows efficient use of the SQL interface in spite of itself.

As with any paradigm change document, several supporting points need to be clarified before the above statements make sense. Here they are:

1. **MPbase** runs on "naturalized data." This is a much more powerful structure for processing the information than normalized data in an RDBMS.

2. SQL was specifically created as both the DDL (data definition language) and the DML (data manipulation language) for normalized data in an RDBMS. It is highly specific to the assumptions and limitations of both.

3. The full functionality of SQL assumes the structures and limitations of both a traditional RDBMS and normalized data. It was designed to try to overcome as many of the limitations of this environment as possible.

4. SQL is a programming language. VERY few end users "code SQL". Most end users enjoy some form of GUI. As an example, in Microsoft's Access the user interface is a point-and-click GUI. After you have defined your query, you have the option of looking at the resulting SQL. Some advanced users(programmers) might even modify it. However the primary end-user interface is the GUI, not the SQL.

5. SQL is both the most common and standard program/database interface language. However, in almost all cases, the interface is a small and well defined subset of SQL. **MPbase** has absolutely no problem with such defined subsets, and therefore, can be used directly as a plug-and-play replacement for the database supporting such systems.

6. **MPbase** can appear to be relational. It does not, however, share the same limitations you would find in a traditional RDBMS. Any question that could be asked using SQL can still be asked and answered using SQL. However, a lot of the complexity of SQL is not necessary when you are allowed to assume the ideal table configuration for your query. This creates a situation where a large portion of the SQL language just does not apply, and so is not supported.

7. DBA's spend more than a little of their time fixing SQL code created by programmers. Any good DBA does not allow the types of SQL constructs that create unnecessarily complex or database intensive queries. This is precisely the type of SQL, not supported by **MPbase**.

8. In **MPbase** the database view allows perfect "virtual tables" to be presented to any SQL interface. This means there is no need to join several tables together to get the result set. All that is required is to request the needed "virtual rows" from the ideal virtual table. This moves some of the complexity of the SQL interface down into the DBA's domain. After all, they are the ones who are supposed to know the data and database.

9. **MPbase** supports (with any appropriate view) SQL, which thinks it is doing a complex summary and/or join. It will simply ignore the sections of the SQL statements not relevant to the efficient and complete resolving of the query. This compatibility feature allows **MPbase** to replace complex RDBMS requests with simple **MPbase** requests from an existing system in a transparent manner.

10. The bottom line is that SQL is part of the problem, not part of the solution. **MPbase** has included sufficient SQL support to allow systems that need to communicate in SQL to continue to function. **MPbase** will provide efficient support of these inefficient interfaces. The **MPbase** view, which is an extremely enhanced version of the traditional database view, allows efficient use of the SQL interface in spite of itself.

This represents the best possible bridge between the two database paradigms. It provides the highest possible efficiency and flexibility while at the same time supporting the current standards in the best way possible.

MPbase And Application Complexity

Simple information access

Here is a common performance assumption Open Sky Technologies would like to challenge: "My database is running just fine. It is my applications that need speeding up."

An information processing system is not arbitrarily split into these two pieces. Every application designer makes information-handling choices. Choices about which functions will be performed inside the database and which will be performed inside the application. The line between the two systems is set by choice.

This choice is based on the current database's performance characteristics. What a database does well, or not well, will be a major factor. Also effecting this choice is database performance. Different application systems will make and use different choices. In most cases the application will attempt to compensate for any lack of information-handling ability in the database.

What this means is that the application will assume database functions as needed to allow the system to work. This creates an environment in which the database is not the primary information-handling system. In this environment the applications become more and more "database like." In short, the database problem becomes an application problem. Hence, the slow-running application is a DATABASE problem.

In the ideal information-handling environment, the application is nothing more than a graphical user interface (GUI). The original purpose of database technology was to provide the ability to get answers from a data store. What most database systems have become are data extract engines. In this mode, the applications must assume a large share of the information-handling requirement. This results in large and complex programs.

When the database is capable of both the needed functionality and the needed performance the applications can become a very thin and simple layer. This layer need only deal with the actual user-interface issues. All of the information-handling issues can be found in the database. This allows far better performance and more consistent functionality than is possible using a conventional DBMS.

In the current environment, it is to a large extent up to the application programmer to dictate access strategy to the database. Doesn't it make a lot more sense to just ask the question and allow the database to control how the answer is found. In which part of the system would you expect to find the best understanding of what is stored and how best to handle it. In short, it is time to let the database do what it was originally intended to do.

So the statement: "My database is running just fine. It's my applications that need speeding up," becomes: "My applications are as fast as possible. Now my database controls system performance." Add this to "I am using the fastest database available," and you really can have a winner.

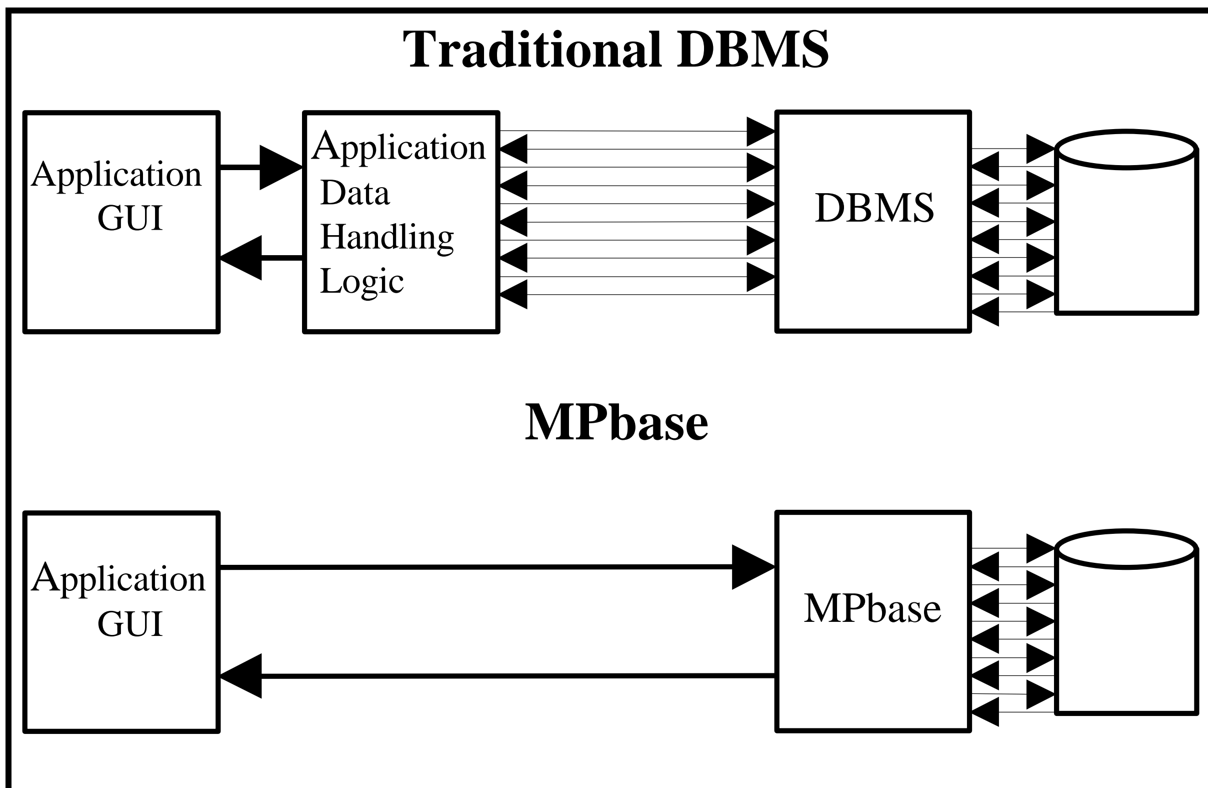
Why MPbase Speeds Up The Whole System

Information warp drive

Why does **MPbase** speed up the whole system? The answer has two main parts. Part one is the environment external to the DBMS. In this part, the application's Graphical User Interface (GUI) first asks a question. This question is translated into a number of database calls. Each call to the database may result in one or more disk accesses. Once the application-data-handling logic has assembled the answer, it is presented back to the GUI.

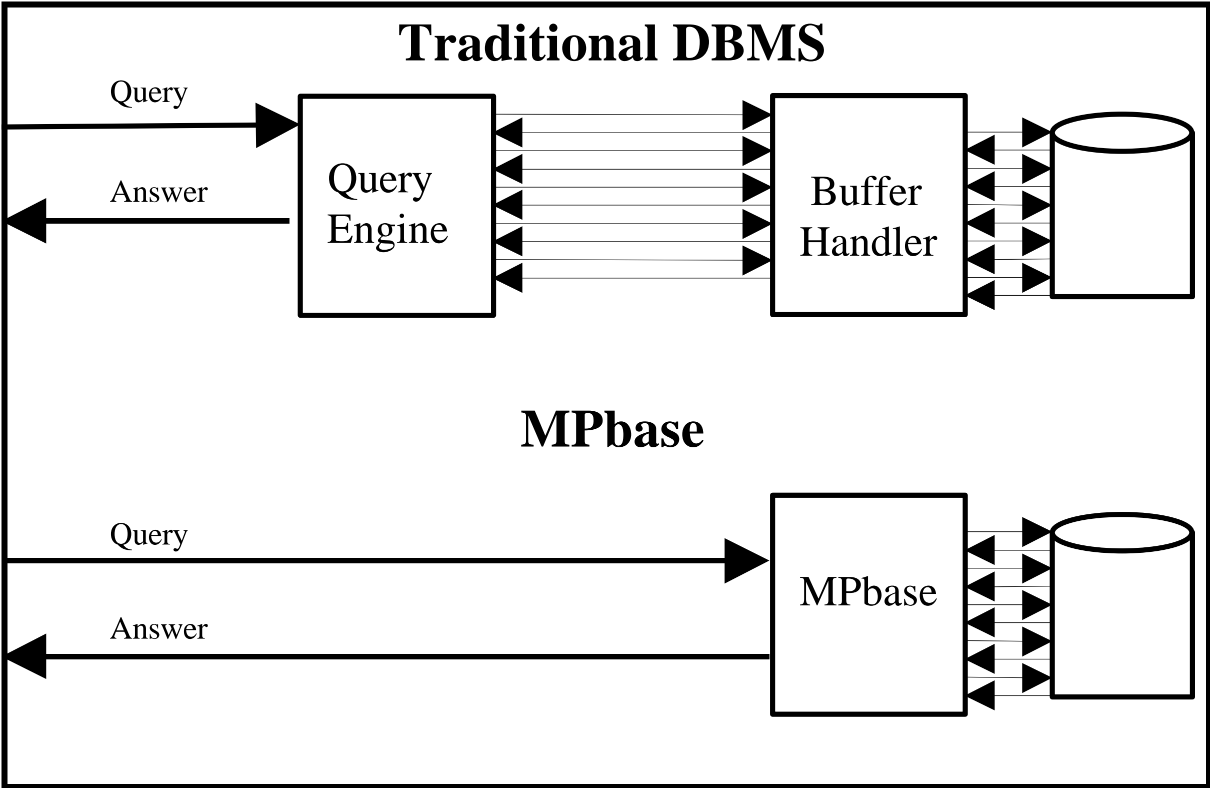
With **MPbase** the question is passed directly from the GUI to the database engine. **MPbase** then makes a number of disk accesses to answer the question. The answer is then passed back to the GUI.

The resulting reduction in communication overhead produces a very significant savings. This combines with the ability of **MPbase** to resolve the query in a more straightforward manner. A manner not possible under the restraints of the more traditional interface.



The second part of the answer is internal to the DBMS itself. For each query to the DBMS the above pattern repeats internally inside the database. In the case of **MPbase** the above pattern is again repeated internal to the database.

This results in even more reduction of communication overhead combining with even more intelligent ways of resolving the query.



The reason **MPbase** is so much faster should be obvious from the above diagrams. **MPbase** can answer the same questions/queries with much less effort. Two reasons this is possible are data naturalization and content addressable memory. The classic relational model does not allow for this type of optimization. It is therefore not possible under a conventional RDBMS.

Naturalization vs. Normalization

The only logical choice

The process used in preparing data to be loaded into a relational database is called normalization. This process involves taking a single-source file and breaking it up into several related tables. The goal for each table is to contain only those data fields (columns: in RDBMS terms) that reference "the key," "the whole key" and "nothing but the key." In addition, repeating fields are forbidden. This results in queries normally needing to join several tables to produce a useful result.

The process used in preparing data to be loaded into **MPbase** is called naturalization. This process involves taking several source files and combining them into a very small number of tables. The goal for each table is to contain all of the data fields that relate to one natural order. This results in queries normally needing to subset one or maybe two such tables to produce a useful result.

While the rules defining relational databases can be beneficial to users and programmers. They are a nightmare to a DBMS's internals. **MPbase**'s solution is to present the user with a de-coupled view of the data. What the user sees has very little to do with how the information is physically stored. This allows "views" of the data to look, act and feel relational, even though the database internally is not.

As an added advantage, the same **MPbase** can have different views supporting different applications and users. These views can be "relational," "hierarchical," or "multidimensional." In addition, the views can support different data formats and communication protocols. So an MVS mainframe may be looking at a hierarchical view in EBCDIC from TSO, while at the same time a Windows PC is seeing the same information as relational ASCII from a Visual Basic application.

The process used to prepare data for loading into **MPbase** is unique to computerized information systems. It most closely resembles the process used by file clerks in the days when businesses were run with people and paper. Through a set of statistical functions the process attempts to find the most natural order for a given set of data. This ordering allows a natural segmentation of the data into a massively parallel framework.

At the same time this process provides, as a by-product, information about any data that does not fit into the natural order. For example, this represents all of the data in the tails of the classic bell curve. This information is invariably one of two types: data in error or interesting data. In either case, this is data you need to know about.

A quick example is Open Sky's phone book demo. Start with 50 files, one for each state. Create two linked tables, one in natural order by name, the second by geography. Produce 80,000+ physical files supporting the two tables. Now, what you have is one logical table viewable as any type of database that makes sense to your application.

This naturalized data would produce incredibly inefficient use of a conventional RDBMS; however, in **MPbase** this same structure is extremely efficient. In the phone book example above, the space used went from an ASCII comma-separated value (CSV) size of 12.3 gigabytes to an **MPbase** size of 1.7 gigabytes. It is important to remember that small size is a byproduct the goal is speed.

The last example applies to data extracted in a denormalized form. While helpful, it does not directly address the issue of what happens to table count when converting from a traditional RDBMS to **MPbase**. A better example in this case is the TIGER mapping database. This is the U.S. census bureau's mapping database. The document describing this database is over 270 pages. The data comes in zipped archives by county.

The database is in a normalized form containing 17 tables. The first step was to fully de-normalize the data. This produced just two ASCII CSV files. These files were then naturalized, producing one table and two indexes. This is supported by only three physical files for each county. Seventeen files with over five Mbytes of data produced three files of less than 400 Kbytes total.

It is generally understood that any query needing hundreds of joins is inherently slow and resource-intensive. **MPbase** overcomes this issue by drastically reducing the number of logical tables in a database. At the same time naturalization provides a substantial performance boost over and above the reduction in join-processing. When added to the decreased storage requirements, this presents the best possible solution for any large or complex system.

Boxes or Attractors

Why not use relational?

At the core of **MPbase** is the unique way it organizes information. This organization is very different from a traditional DBMS. In a traditional database the various tables, rows and columns can be viewed as boxes. The access to each box is by its label. The more precisely defined the label, the less you can put in one box. So you can either have lots of precise little boxes or a few big, but vague, boxes.

The problem is that neither design makes a good general choice. This would tend to imply that the best choice would be the midpoint between the two. The unfortunate reality is that the midpoint suffers from the problems of both while having few of the benefits of either. It really is the worst of both worlds.

This is what causes high-performance database designs to specialize in only one type of transaction. This is also what causes the more general-purpose database designs to be tolerable for anything, really good at nothing. Any attempt to improve some single aspect of the general-purpose database design will tend to trash some other area of functionality.

This either/or mentality is so ingrained in the current database paradigm that no one dares to ask "What if we could have both?" As long as you are predefining boxes to hold your data you cannot. If you were to give up the fixed boxes, how on earth would you ever find anything? The answer is to let the data sort itself out. This is only possible under a content-addressable schema.

Using this content-addressable model, the need for fixed metadata goes away. If you do not have fixed boxes you do not need fixed-box labels. What you get in its place is virtual metadata, or the ability to define and label your boxes on the fly. By definition, if the data content determines its location in the data store, the more alike the data, the closer together it is stored.

In this model you could say: like data attracts. Hence the most common data elements are the best attractors. This creates the most natural possible structure for loading, updating, finding or analyzing the data. It allows the ability to "zoom" in and out. From a high-level macro view to a detailed analysis of subtle differences between almost matching data.

Just looking at the resulting structure can be very informative. Data found in large clumps is going to be very similar. Data found off by itself is normally going to be ether in error or interesting. Some types of analysis will focus on the "stray" data, others on one or more of the data "clumps." In ether case the other type of data can easily be kept out of the way.

Data Types in MPbase

Almost anything goes

MPbase can store any class and type of data a computer can contain. Data classes include tabular, raster image, vector image, spatial, textual, attribute-value pairs, etc. **MPbase** does not treat all data classes the same. A data class is stored in its natural order. Each class of data has a different kind of "natural" order. This means using different approaches to "naturalize" different classes of data.

In addition to differences in data class, there are also differences in data types. Data types include integer, text, floating-point and binary, in addition to several mixed forms. Within **MPbase** the data typing can be either hard or soft. In the case of hard typing an update will fail if the data does not conform to the expectations. With soft typing any data can be put into any field or column.

Soft typing can be used to allow "raw" data streams to feed a database directly. Editing can then be a second step. This allows editing of the data in context. This also allows the roll back logs to see the original data when required. With soft typing the storage efficiency for "out of type" data is not very high. However, this is a small price to pay for the enhanced editing and the ability to include or exclude it from queries.

The efficiency with which **MPbase** stores any one class and type is dependent on the data's information content or entropy. This entropy value can very widely within a database. However, for any one class it will tend to average into a relatively small range. As an example, for a scanned photo requiring lossless compression one should expect around 50%; for tabular data between 80% and 99.9%.

It is important to note that **MPbase's** compression works significantly better on "real" data than on generated test data. This is due to the relative entropy content of random test values vs. their real world counterparts. Part of the naturalization process involves taking a statistically significant portion of the real data and analyzing it. This allows **MPbase** to fine-tune the type of encoding it will use to store the data. This encoding may vary over time without any need to reformat the older data.

In addition to the standard types, **MPbase** can support special custom classes and types. These types then become part of **MPbase**. One example of this special type is "address number." This is a mostly numeric field. However, it also efficiently supports n, s, e, w, -, / and any trailing letter. This means that "12w345-1/2" is an "in type" value. In addition this is a soft typing and so allows any "out of type" values as well.

So, as far as data typing and **MPbase**, almost anything goes.

Removable Media

A better choice than you might think

MPbase often makes it possible to attain “hard-disk-like performance” with the use of less costly removable storage media. This allows the optional use of optical or, in some cases, even tape as the storage media for a database.

There are several aspects of **MPbase** that make it ideal for any database or data warehouse using fixed, removable, or mixed media. First, the process of naturalizing the data creates a segmented sequential internal layout. Second, by working with the file system directories, use of cache is maximized. Third, due to the parallel nature of **MPbase** the effects of wait times are minimized. Fourth, the compression both reduces the amount of data transferred and increases the speed at which the search executes.

The process of naturalizing data creates a productive layout for removable media. **MPbase** breaks the information up into lots of discrete files. These files are in groups such that one or two large blocks will tend to resolve most queries. The placement of data in the naturalized format tends to minimize the number of such blocks needed by any one query. And, **MPbase** at the same time reduces the number of disks or tapes needed by any single query.

The indexing is to a large degree resolved by the file system index. This index is almost always in a memory cache and is at least partially loaded during a mount. In the case of media management libraries, this information is normally kept on disk. This allows a very quick selection of the sections of the database that will need to be searched.

As the **MPbase** is massively parallel it can be working on many queries or sub-queries at the same time. This reduces the impact of media waits. With many parallel internal paths the searching of blocks already in memory occurs in parallel with the media waits for new blocks.

When added to the compression, which makes the system more CPU rather than I/O-bound, you create the best possible scenario for a database containing removable media. Keep in mind that this system is NOT a waster of CPU time. It reads the data smart, not hard. The structure of the naturalized data allows it to spend most of its time on useful work rather than on reading and parsing unnecessary data blocks.

Although each of the above will help with removable media, all of the above have a tremendous effect on the total throughput.

Year 2000 & MPbase

What is and what is not

There are both **MPbase** assumptions and database issues with Y2K.

First the assumptions (over which the **MPbase** software has no control):

1: The hardware and BIOS are compliant.
2: The operating system and its file system are compliant.
3: The "C" library date functions are compliant.

Beyond these 3 assumptions **MPbase** has no internal dates or times that are not fully Y2K compliant.

Database issues:

As with any database **MPbase** can only store what it is given. A non-compliant system *could* be created under **MPbase** (or any database for that matter). This is of course ***not*** recommended. When being loaded with incomplete date information **MPbase** can automatically fill in the century digits of the year ***if and only if*** **MPbase** is provided with a valid set of assumptions about which century to use. As with any database **MPbase** has no way to validate such assumptions.

A non-compliant database can be loaded into **MPbase** and then corrected in-place without reloading. This again assumes a valid information source or assumption is available to determine century. Given **MPbase's** speed, it is normally faster to move the data to **MPbase** & convert it, than to just convert it in a current database.

MPbase Architectural Limits

Why have limits?

MPbase, due to its unique architecture is not bound by the traditional limitations. However, it is all too easy, to say this product has no limits, so as ridiculous as they may be here are the **MPbase** architectural limits.

Size Limits:

Maximum database size:	The total storage capacity available to the cluster.
Maximum logical row size:	Database size.
Maximum column size:	Database size.

Number Limits:

Maximum nodes in a cluster:	The total number of machines available.
Maximum nodes in a database:	The useful limit varies but for the phone book demo (95 million rows) it is 80,000 nodes.
Maximum number of rows:	Database size/**average compressed** row size.

Scaling factors:

For nodes:	Linear+ performance to maximum cluster size.
For disks:	Linear capacity to maximum database size.
For network connections:	Linear performance/capacity to max cluster size.

Disk space requirements:

Data space multiplier:	* 0.5 to * 0.005 normally about * 0.1
Software space	less than 1 Mbyte.

OS requirements:

Server:	Any UNIX supporting TCP/IP.
Client:	Any supporting TCP/IP.
Single box: (client, server & application)	Any supporting “C”.

Memory (RAM) requirement: As little as will run the OS (less is better).

Downtime requirements:

None.

MPbase can be moved to a new set of hardware or even a new site and never needs to be taken down. Backup/restores, reorganizational reloads, …

There is no *requirement* for MPbase to be taken down ***EVER.***

Product Plan & Deliverables by Phase

Where we are & where we are going

The Final Goal

MPbase can eventually be a wholly new, full and complete commercial DBMS product. Toward this end it will have all of the needed infrastructure and support such a product requires. It will be the most powerful, fast and efficient DBMS on the planet, bar none. **MPbase** will be an environment capable of supporting all aspects of IT functionality. It will be *the* way to handle information in the future.

MPbase will be the first fully automatic data intelligent information handling system. It will only need to be shown a sample of the data and a terse set of field definitions to construct a database's internal structure. There will be no need for the traditional DBA role. The system itself will perform the DBA functions in ways not possible through a human/machine interface.

The **MPbase** views will be controlled by the users and application programmers. There is no direct connection between views and the physical storage. This means there is no need for the usual DBA negotiation between the different systems and users. The internal **MPbase** structure will be self modifying. It will automatically change over time, based on changes in the patterns of the data. In addition, tuning will be automatic, based both on the data and on observed access patterns.

In order to release **MPbase** as a user-implementable product, it will be necessary to evolve **MPbase** through several additional phases. At this time the most important development phases, those that represent its extraordinary proprietary advantage, are already completed and proven. The root technology that is **MPbase** is well on its way toward its final goal as a complete DBMS package.

The phases of previous development are:

Phase 1. Realize the problem has a solution. *~1983*

The first step for **MPbase**'s technology, was to realize that there was no efficient way to work with information within the current information processing paradigm. Before a rule changing approach can be developed, it must be discovered and realized that the old rules are inadequate.

It is not sufficient to think: good theory, bad implementation. The realization must occur that the theory itself is flawed at some base level. There are currently a lot of good information processing ideas. These ideas are handicapped by the foundation upon which they are built.

It is not enough just to find the problem. There are a lot of unsolvable problems out there. One must realize there is a solution. It does not matter that you may not know how to make use of it. It must be known to exist.

Phase 2. Make it work for one case. *~1989 - 90*

The next step is to find a way to make the solution work. This starts with one specific case. In the case of the **MPbase** technology this was an imbedded system. A program "imbedded" in a specific piece of hardware.

Phase 3. Make it work for more cases. *~1991 - 92*

Once the initial solution is working it is time to stretch the envelope. Time to start pushing the limits and find out how good or bad the solution really is. This is where niche solutions are separated from potential paradigm shifts.

Phase 4. Demonstrate at large scale. *~1996 - 97*

For **MPbase** this was a proprietary purpose-built database. It was initially loaded with over 50 billion row of information. It sat on a cluster server of 30 machines. Extract performance was over 90K rows per second. It was at this point that a lot of the new theory was tested and verified.

Phase 5. Develop initial technology deliverables. *Available today as purpose-built version*

1. DBA expertise & tools:
   to design initial purpose-built versions of **MPbase**.
2. Development expertise & tools:
   to allow quick development and integration of **MPbase** systems.
3. Conversion expertise & tools:
   to allow quick and easy conversion of even the most massive data stores.
4. Cluster management expertise & tools:
   to allow easy setup of **MPbase** fault-tolerant parallel-cluster servers.
5. Parallel hardware I/O architecture (disk, data & user nodes)
   Design, implementation, & testing
   to provide guaranteed performance of **MPbase** implementation.
6. Tuning expertise & tools:
   to facilitate the initial **MPbase** shakedown and validation phase.
7. User education & operational tools:
   to allow ongoing support and maintenance of any purpose-built **MPbase**.
8. Ongoing support & enhancement:
   for **MPbase** available as requested from Open Sky Technologies.

At this time **MPbase** is only available in a purpose-built version. The only real difference between this and the final version is that the DBA & tuning functions still require our staff. After a database is initially set up, there is no practical difference.

Phase 6. Find initial problems to solve with purpose-built version. **Target** *1998*

The next step is to find a good set of initial problems. Problems that are isolated enough to allow the relative freedom required with a purpose-built solution and preferably solving problems inside of other products. These would be the type of problems with high return to offset the purpose-built effort.

Phase 7. Generalize the solution. **To do**

At this point there has not been a large enough set of implementations to create a truly general version. Even though Open Sky has quite a large knowledge base with this technology, there is not quite enough yet to create the shrink-wrapped box version. The generalization will need to be done in three parts.

The first part will be to polish the tools and assemble them into the final "kit" forms. The second part will be to create an Artificial-Intelligence (AI)-based virtual DBA as an additional set of internal functions and tools. The third part will be to formalize the needed administrator knowledge into a fixed-class format.

Phase 8. Final product deliverables. **To do**

Most of this list is currently in some partial form. To complete this list will require the additional information we will be acquiring during phases 6 & 7.

1. Core software

2. Table builder - AI tool set
3. Monitor/tuner - AI tool set
4. View creator - tool set
5. Operations interface - tool set
6. Cluster management - tool set

7. Pre-configured cluster hardware
8. Turnkey, hands-off server systems.

9. VB access samples
10. C access DLLs & samples
11. HTML access samples

12. Basic system administration class
13. Operations command & control class
14. Advanced system administration class
15. Getting the most from… developer class

16. Remote console operations support (instructional)
17. Remote console operations support (active control)
18. Consulting support
19. Hotline support

Building The Phone Book Demo

This narrative will walk you through the steps taken while creating an **MPbase**. It will follow the creation of the phone book demo. Although the specifics will change based on the data to be used, the general flow will remain the same.

1. Select the data to be used.

   The 1996 addition of *ProCD, Inc's*, select phone product was picked. There were three main reasons for this choice. First, their product allowed unlimited downloading of the data into any database. Second, it contained 95 million rows. Third, it was data that would be recognized by anyone.

2. Naturalize the data.

   A) **Determine what physical tables will be needed.**

   For the phone listings it was decided that two physical tables would be needed. All the information relating to phone number and address would be in the one table. The other table would contain the name and business (SIC) information.

   B) **Determine a natural order for each table.**

   The order for the first table was set by State, Sorting center (first three digits of zip code), Town name, Zip code, Area code, Street name and finally Address. This sequence most closely approximates how you would physically find this data outside of computers in the natural world.

   The order for the second table was set by State, First letter of name, Sorting center then rest of name. This sequence was influenced by the need to link it to the first table. The most natural order for the second table would have been by full name. However, this would have created the need for a large and unnatural key in the first table. The solution was to link the two tables at the sorting center level. This required "state" and "sorting center" to come before "rest of name."

3. Create the data access layer.

   A) **Determine the groupings to be used**.

   This is where the directory trees to be used as part of the database are determined. The primary goal here is to create 500Kbyte to 1.5Mbyte low level files. A secondary goal is to group the directories in a way that will minimize the impact of any less than fully normal tables.
   The directory tree for the first table is table/state/sorting center/file
   The directory tree for the second table is table/state/first letter of name/file

   B) **Create the table-specific access-layer code.**

   For each table two data-specific programs must be created. One to add and update rows and one to extract rows. This is part of **MPbase**'s "magic." The low-level files are compressed using multidimensional, data-intelligent, run length encoding. The result is a continuation of the naturalized structure down into a compressed file format.

4. Load the database.

   For the phone book demo this took over two weeks. Most of this time was spent extracting the data from the source database. Each state was read and loaded individually. A PC ran the extract to a tab-delimited format. This result-set was then "naturalized" using PERL, C shell, and sort. The resulting set flat files were then loaded with the data access layer.

5. Set up the initial interface

   Two Visual Basic programs were then written for this demo. The first to show what can be done in terms of a very powerful interface. The second to demonstrate how little is really needed to connect to an **MPbase**.

Phone Book Demo Statistics

Sizes

Number of rows	95,153,940
Fixed length row size	281
Fixed length total size	26,738,257,140
Average CSV row size	129
Total CSV size	12,274,858,260
Average compressed row	17
Total compressed size	1,669,172,000

Speeds

Benchmarks were run on (1) a single 170MHz turbo SPARC, with 16Mbytes of RAM and two disk drives, (2) 4-400MHz Pentium IIs with 64Mbytes of RAM each with two disk drives.

	1-SPARC	4-Pentium IIs
Single thread search	58,340 rows/sec	(1-CPU) 158,589 rows/sec
Four up search	71,329 rows/sec	(4-CPU) 660,791 rows/sec
Naperville IL	40,911 rows in < 1 sec	40,911 rows in << 1 sec
Chicago IL	728,302 rows in 13 sec	728,302 rows in 5 sec
All of Illinois	3,651,092 rows in 58 sec	3,651,092 rows in 14 sec
All of USA	95,135,940 rows in 1345 sec	95,135,940 rows in 144 sec
Main St USA	1,155,266 rows in 627 sec	1,155,266 rows in 29 sec

Real World Numbers #1 "Data Transfer"

A 7/24 batch data transfer system

A good example of **MPbase**'s parallel fault-tolerant architecture is a data transfer system set up by **MPbase**'s author. This was done long before **MPbase**, but is a good example of this type of paradigm.

This system made use of "spare bandwidth" on a T1 to move data from an IBM ES9000 to a cluster of UNIX servers. It replaced the need to run a van with a load of 9 track tapes over 300 miles twice a day. In addition, these tapes could not be reused after leaving the data center, due to operations' policy. The other bonus was that files larger than one tape could now be used.

Statistics:
- 1.5+ terabyte each year,
- 7/24 darkroom operations,
- Self pacing to 50% of the available bandwidth during day,
- Used 98% of the available bandwidth at night,
- Highly fault-tolerant multi-path transfer,
- Automatically supplied one BBS and three departments with data,
- Controlled by mainframe naming-convention.

After being asked several times an hour about link status, a console was set up facing the window in the server room. After this, there was frequently a good size group watching their files move through the system.

An amusing note:
Most of the users of the system were not fully aware of just how robust the link was. During one large and critical transfer I was asked by the user to take an early lunch. He was concerned that other work might somehow interfere with the transfer. My response was to walk over and shut down the primary machine performing the network routing into the server.

After a moment of total panic the transfer resumed by routing through a secondary network connection. I explained to the user that there were three such links and all would need to fail at the same time to interrupt his transfer. For some strange reason I was never again asked to take an early lunch due to an in-progress transfer.

Real World Numbers #2 "Big DB"

A big fast database

Here are some numbers from a recent custom version of the **MPbase** technology.

Operations:

- 7/24 operations, no need for any downtime,
  not even for OS upgrades, reorganizational unload/reloads or hardware fixes.
- "Dark room" hardware mode,
  no one ever needed to look at the cluster except to replace broken hardware.
- No single point of failure,
  any switch on the cluster could be shut off and ***not*** bring down the database.
- System feeds MVS, UNIX, Windows and HTML (Web access),
  at same time from same data, EBCDIC, ASCII, straight or swapped byte order.

Hardware:

- 30 - 110Mhz SPARC 5 servers (3.3 Bips total),
- 390 - 2.1 Gbyte 7200 RPM SCSI disk drives. (819 Gbytes total),
- 30 - 40 Mbytes/sec fast & wide SCSI 2 channels (1200 Mbytes/sec total),
- 30 - 20 Mbytes/sec fast SCSI 2 channels (600 Mbytes/sec total),
- 30 - 100 Mbit/sec fast ethernet segments, switched (375 Mbytes/sec total),
- 5 - 10 Mbit/sec ethernet segments, switched (6.25 Mbytes/sec total),
- 5 - 7 foot high cabinets (only 20 square feet of floor space).

Data:

- 50+ billion records (initial load),
- 200+ billion records (predicted final size),
- 93% compression from 52 byte transfer format,
- 85% compression including the mirror copy of the data,
- compressed data 320 Gbytes with mirror,
- uncompressed data 2 Tbytes for one copy.

Software:

- 126 executables: 74 C shell scripts, 33 PERL scripts, 17 C programs,
- 1500+ automated daemons with queues.

Performance:

- Internal validation rate - 16 billion rows/hour, 267M rows/min, 4.4M rows/sec,
- Sequential scan rate - - - 16 billion rows/hour, 267M rows/min, 4.4M rows/sec,
- Random extract rate - - - 90K rows/sec at a 25% hit rate (400K keys per second),
  6M rows/min, 360M rows/hour, 8.6 billion rows/day,
- Max load rate (ext) - - - - 8.6 billion rows/day, 360M rows/hour, 6M rows/min, 100K rows/sec,
- Unload/reload rate (int) - 24 billion rows/day, 1 billion rows/hour,16M rows/min, 277K rows/sec.

Inventor's Notes on Paradigm Shock

No easy path

Lets start with, "Who is Randall Nelson, the creator of **MPbase**?" I am a computer speed freak. I am someone who has made a career out of making systems run faster. Someone who at one time or another has been a DBA (database administrator) for most commercial DBMSs: M204, IMS, DB2, Oracle, Sybase, IDMS… Someone who has both worked on and administrated a wide number of different systems. Everything from IBM mainframes to standalone imbedded systems code on a 4.77MHz XT. Operating systems including: MVS, DOS/VSE, VM/CMS, UNIX, CPM, DOS, Windows…

Many years ago I had an eye-opening experience that started me down a different road. At the time I was the DBA for an IMS database. We had just converted an AR ledger from punch cards to an IMS database. It took several hours to run the AR ledger total. I took this total to the head filing clerk to verify.

I assumed it would take days to duplicate what the computer had done in hours. So, I started to walk away. As I turned she said "Wait a minute and I will check this for you." I almost fell off my feet. I watched as she used some files, a set of 3x5 cards and an adding machine to produce the same number in minutes not hours, much less days!?!?!?

Up until this point, I had been convinced that the computer was the fastest way to handle information. There was no way to do this kind of work using files and paper. I realized just how little I knew about working smart, not hard. It took over 20 years to fully develop the computerized form of that "minutes not hours" information handling system. This resulted in **MPbase** a "seconds not hours" information handling system.

What you are about to read (or just read), sounds like overstuffed marketing hype. This is the result of a paradigm shift "catch 22." If you explain a new paradigm using the old rules and assumptions you will prove yourself wrong. If you explain using the new rules and assumptions you sound like a babbling idiot. Having said that, let's try an example.

Let's try to explain a gang (linked set) of six diesel locomotives to someone who thinks only in terms of steam engines. Under the old rules this is simply six engines used together to pull a single train. For someone working in the old paradigm there are several issues why this will not work. Issue #1, you could never balance the steam pressure. Issue #2, there is no possible way to keep several sets of controls in sink. Therefore this can not work. Two engines maybe, but six no way.

Under the new rules the old issues do not make sense. With the new rules, several engines provide power to lots of little motors. This means that whether the engines and motors are in one physical locomotive or several is not relevant. From the steam point of view this is utter nonsense.

The more knowledge one has the harder it is to cross the bridge to a new paradigm. I was no exception to this rule. During the years this paradigm was being developed and polished there were several times I had to stop and ***re***-convince myself of its validity. There were times during testing when I would look at what I had running and decide that what I was seeing just ***could not be***. Each time I had to devise some way to prove to myself that I was not nuts. Here is one example:

At one point during an early test, I just stopped believing what the system could do. The only solution was to spend over 4 hours single-stepping the code to convince myself that it was really looking at every single row it claimed to. This was logically a waste of time, as counts and extracts precisely matched the results derived from the flat files used to load the system. Adds, deletes, and changes were correctly reflected in the output.

In short, I ***knew*** what it was doing. I would not ***believe*** what it was doing. I had all the needed proof, but I still could not except the results. This was my personal version of "I will see it, when I believe it."

I would ask you to suspend your disbelief long enough to allow yourself to get past the proof and on to the belief. I do have a running demo that I will show to anyone. If you are shown the demo, I would like you to be able to see it.

Evaluating New Technology

Notes on paradigm testing

When faced with a potential new paradigm the first thing you must consider is by what criterion will you judge it. It is frequently the case that using the tried-and-true technological yardsticks will give you false or highly misleading results. This is due to how technology yardsticks are created. Lets take a look at this process.

The typical development of yardstick criteria follows a predictable pattern. The criteria start as real world numbers, for example, this job runs in X hours, or this engine has Y horsepower. But these are hard numbers to generalize; they do not compare well across multiple systems and companies. They depend on too many variables to be reliable and repeatable. The solution to this problem leads to the next step. This is to find "low level" numbers that seem to be good predictors of the "real" numbers.

Here is where the yardsticks tend to get paradigm-specific. These "low level" numbers can be excellent predictors in one paradigm and terrible predictors in another. Numbers like, this job moves Xk per second from disk, or this engine has Y cubic inches. These numbers are NOT direct performance numbers. They are predictors of performance. In the case of the cubic inch, this predictor works fairly well until you look at the rotary engine. It fails completely when used with turbine engines. Both rotary and turbine engines represent different engine paradigms.

It is very important to understand the difference between these "real numbers" and the "predictors." Here lies the major problem in evaluating any potential new paradigm. Given the invalidation of the prevailing yardsticks, how can anyone ever evaluate a truly new technology? How can someone hope to develop a fair, generic, and paradigm-free yardstick? The first step is to separate the "real numbers" from the "predictors."

Toward that end, I will now present several common computing predictors. For each I will provide an explanation as to why they are truly predictors and not always valid.

1: Percent of CPU utilization,
In the current paradigm a low percentage equals higher total throughput. This is a good thing. In other paradigms, high percentages equal better throughput. The "real number" for this is total CPU time consumed, and the lower the better. Be careful because this is referring to total system CPU consumed. It is a very hard number to quantify on a system running more then one task. There is a lot of CPU overhead at the system level. A significant part of this is not reported at the single-task level.

2: I/O latency,
I/O latency, commonly called "disk wait." In the current paradigm, the lower the number the better. In other paradigms this number can very from having no effect, to being in the higher-the-better category. This wait represents the only free time the system has to work

on other tasks. In some environments this is a good and necessary thing. In a parallel environment, the disk waits can be "free" as they do not consume CPU time.

3: Communication latency,
This is the wait associated with any communication task. In the current paradigm, lower is better. However, several of the current paradigm tricks used to produce lower communication latency will increase the system overhead. This has the effect of reducing total system throughput. In short, there are times where increasing this wait time will increase total throughput. In other paradigms, increasing this value's maximum can have the effect of reducing the *average* wait.

The only true measure of computer performance is total system throughput. Numbers like: this job, on this hardware, runs in X minutes. Or results like: this system, on this hardware, can support Y thousand users before response time goes over 1.5 seconds. Both of these examples produces an apples-to-oranges situation when comparing to any other system. This unfortunately makes any comparison tricky at best.

So how can any meaningful comparison be done at all?

To some extent this needs to be evaluated on the basis of what has meaning to you. The strategy I would recommend is as follows:

1: Identify your business make or breaks.
This will allow you to "fix" some of the variables. Each number you can fix reduces the complexity of the comparison. Be careful that you use only real world business numbers here. System numbers, "predictors" or otherwise, have no business here.

2: Clean up the fixed business values.
Once you have a set of fixed numbers, it is time to make them paradigm-free. Numbers like: from the time we receive the last update, this processing must complete in X hours, are good. Numbers like: this batch run must complete in Y hours to allow time for the index-build, are bad. Real world business numbers only.

3: Equate values that are not fixed.
Once the fixed values are set the next step is to equate as many other performance values as possible. A good way to equate hardware platforms is to use processing cost. Once the fixed-business-numbers are met the best system has the least total cost.
Figuring the cost of a CPU second can be eye-opening. The cost of a "CPU second" = ("Total system cost, including supporting sub-systems" / "Useful system life in seconds") + ("Total operational cost per year" / "Number of seconds in a year"). Once you know the cost of a CPU second you can derive the cost of running your task on each system. If you do this for both of the systems being compared, you may find one system is less expensive even when it is slower and occupies more physical boxes.

4: Remove issues that are not business based.

> Internal system architectures have no place in the evaluation. What internal model a system uses IS NOT RELEVANT to how it will perform the needed business functions.
> If you think that this is the case you are by definition stuck in the old paradigm. A new paradigm is by definition a new way of accomplishing some task. No large benefit can ever be found by doing the same old thing "a little better."

When you use this approach to comparing systems, you are in a position to take advantage of the really big improvements. Paradigm changes can be evaluated using this simple and sane approach. You can decide based on business facts which system can really meet your requirements best. You will finally have truly the best system for the business need.

Reasons To Use MPbase

1. You need data access performance with no architectural top end.

   Your initial system is small but you may need to scale quickly and there is no telling how much performance you may need later.

2. You need extreme reliability and do not wish to buy an IBM Sierra or Tandem.

   Your life would be easier with a 99.9999% uptime database. But you can not make the business case for big budget hardware.

3. You have, or would like, a *network-centric* environment and need a network database.

   You can never fully achieve a network-centric environment with a conventional CPU-centric database. Until all your platforms have equal access to your data you will remain locked into a CPU-centric paradigm.

4. You need to move or convert a "must-be-up" legacy database and do not wish to acquire a second set of mainframes for the process.

   You have a monster database on your mainframes. This database takes everything your system has to keep it running. If you need to move or convert it, there is no way to do it without killing your business.

5. You need a single database accessible from all of the platforms in your shop: PC, Mainframe, Midrange, UNIX…

   You are maintaining multiple copies of the same data so that different systems can all see and use it. If updates are present, this is always a nightmare and a total disaster waiting to happen.

6. Your operational computing budget is putting you out of business.

   When you need to get out from behind the eight ball, you need a change of paradigm. The CPU-centric paradigm can put and keep you in a nasty corner.

7. You want to mine your corporate data, but you cannot spare the processing power.

   The largest data store in the world is of no use if it cannot be processed.

8. You need to get a major project done quickly.

9. Two, Three, or all of the above.

MPbase, FAQ

Forward

Like most database management systems, **MPbase**'s characteristics depend to a large degree on the specific data and access characteristics. However, unlike most database management systems, there is no need to make tradeoffs. Each characteristic can be determined individually and independently. In addition, multiple different options can be selected for a single characteristic. For example, a single copy of a single database can be accessed by one method tuned for On-line Transaction Processing (OLTP), and at the same time by a different method tuned for batch list processing.

1) Q I have an OLTP system supporting 25,000 simultaneous users with a sub 2-second response time. What will an **MPbase** do under similar circumstances?

A If there is no need to improve the above numbers, you could expect the same performance for between 1/10 and 1/100 the current cost. At the same time, you would gain the ability to reduce response time and/or increase the maximum number of users to any level at any time. Given the needed hardware, 250,000 users with sub-second response time would be a reasonable expectation.

2) Q My DASD farm has 24 terabytes of data and costs $24,000,000.00/year in lease, maintenance and environmental expenses. What about yours?

A The **MPbase** stores the data in a highly compressed format. Therefore, assuming your data is in a flat ASCII (text) format, you should expect to need about 300 gigabytes of workstation disk per copy. At 1996 prices, this would be about $35,000.00/year per copy which includes the supporting I/O cluster. Also, this type of disk has no special environmental or power needs. This produces significant savings over the traditional machine room. Note, that at this price, multiple copies are recommended.

3) Q My data store needs 24 terabytes and grows at 15% per year. How does an **MPbase** scale at this size and rate?

A **MPbase** scales by adding "nodes." A node is defined as one processor and a set amount of disk capacity. The ratio of disk/processor is determined by the access requirements. Assuming for a moment that the above system starts with 100 nodes you would need to add 15 nodes per year. There is no upper limit to the number of nodes that can be added to one system.

4) Q I have 24 terabytes of storage. It takes 12 weeks for a full backup and two hours per day for an incremental backup. How would your approach perform under similar circumstances?

A If there is no need to improve the above numbers you would expect the same performance for less cost. However, with the assumptions from the answers to questions 2 and 3 above, if required, a full tape backup could be made in as little as 30 minutes. This would require one tape drive per node (initially 100). In addition, a second disk copy could be kept in sync in real time.

5) Q My system runs 24 hours/day and seven days/week. How would your solution protect against outage?

A An **MPbase** with two or more copies of the data is fault-tolerant at the node level. This allows configurations that have ***no single point of failure***. This means any node may be powered down at any time and still have full access to the data. In addition, large clusters normally have a number of "spare" nodes. These spares can automatically assume the role of any node that may fail. Recovery involves rebuilding the data files from the second online copy. Once this is complete, the cluster is back at full strength, with one fewer spare, naturally.

6) Q What is the incremental cost of a complete disaster recovery site and operation?

A There are two main modes of disaster recovery available to **MPbase** installations. In addition, any combination of these two will also work.

1) Full hot site with the files in sync, real time.

   With this option cut-over time measured is in seconds.

   The cost can vary from the cost of the connecting data links, to the link costs plus the full cost of the cluster. This is based on how well the system can tolerate less-than-full performance. With three working sites, loss of any one site reduces the total capacity by only 30%. If the cluster is designed with 30% extra capacity, the net effect to the users is zero.

2) Store the small set of components that cannot quickly be rented and rent the rest when needed.

   With this option cut-over time is one or two weeks. This depends in large part on the backup/restore option selected.

Cost could be as little as $20,000.00 until needed. The cost to rent the "commodity class" hardware for a full year should about equal the purchase cost.

7) Q I have fifty programmers, systems people, and operators who report to me and make up my domain. What would your approach do to help me significantly reduce costs and still maintain my department's significant contribution to the organization?

A First, **MPbase** can substantiality reduce your costs. This reduction would come from hardware, system software, user software, maintenance, environmental, and operational costs. At the same time, significant new functionality and capacity could be added.

Second, as staff is freed from day-to-day operational tasks, they could be re-deployed to increase the value of your department within the company. In the current business environment with an emphasis on re-engineering, operational units are valued by output not input.