Distrib Parallel Databases (2008) 23: 99–126
DOI 10.1007/s10619-007-7023-y

PnP: sequential, external memory, and parallel iceberg cube computation

Ying Chen · Frank Dehne · Todd Eavis · Andrew Rau-Chaplin

Published online: 3 January 2008

Abstract We present "Pipe 'n Prune" (PnP), a new hybrid method for iceberg-cube query computation. The novelty of our method is that it achieves a tight integration of top-down piping for data aggregation with bottom-up a priori data pruning. A particular strength of PnP is that it is efficient for *all* of the following scenarios: (1) Sequential iceberg-cube queries, (2) External memory iceberg-cube queries, and (3) Parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.

We performed an extensive performance analysis of PnP for the above scenarios with the following main results: In the first scenario PnP performs very well for both dense *and* sparse data sets, providing an interesting alternative to BUC and Star-Cubing. In the second scenario PnP shows a surprisingly efficient handling of disk I/O, with an external memory running time that is less than twice the running time for full in-memory computation of the same iceberg-cube query. In the third scenario PnP scales very well, providing near linear speedup for a larger number of processors

Research partially supported by the Natural Sciences and Engineering Research Council of Canada.
A preliminary version of this work appeared in the International Conference on Data Engineering (ICDE'05).

Y. Chen
Microsoft Corp., Redmond, WA, USA
e-mail: ychen@cs.dal.ca

F. Dehne
School of Computer Science, Carleton University, Ottawa, Canada
e-mail: frank@dehne.net

T. Eavis
Department of Computer Science, Concordia University, Montreal, Canada
e-mail: eavis@cse.concordia.ca

A. Rau-Chaplin (✉)
Faculty of Computer Science, Dalhousie University, Halifax, Canada
e-mail: arc@cs.dal.ca

and thereby solving the scalability problem observed for the parallel iceberg-cubes proposed by Ng et al.

Keywords Pipe 'n Prune (PnP) · Iceberg cube query · Data cube · Parallel computing · OLAP

1 Introduction

One of the most powerful and prominent technologies for knowledge discovery in Decision Support Systems (DSS) environments is On-line Analytical Processing (OLAP) [5]. By exploiting multi-dimensional views of the underlying *data warehouse*, the OLAP server allows users to "drill down" or "roll up" on hierarchies, "slice and dice" particular attributes, or perform various statistical operations such as ranking and forecasting. To support this functionality, OLAP relies heavily upon a data model known as the *data cube* [15, 17]. Conceptually the data cube allows users to view organizational data from different perspectives and at a variety of summarization levels. It consists of the *base cuboid*, the finest granularity view containing the full complement of d dimensions (or attributes), surrounded by a collection of $2^d - 1$ sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. The data cube *operator* (an SQL syntactical extension) was proposed by Gray et al. [15] as a means of simplifying the process of data cube construction. Subsequent to the publication of the seminal data cube paper a number of independent research projects began to focus on designing efficient algorithms for the computation of the data cube [1, 2, 17, 19, 20, 23–28, 30, 31].

The size of data cubes can be massive. In the Winter Corporation's report [29], the average size of data warehouses covered exceeded 10 Terabytes. Perhaps more importantly over the last four years the average size rose 243%, while the maximum size rose an astounding 578%. One approach for dealing with the data cube size is to allow user-specific constraints. For iceberg-cubes (e.g. [2, 11, 30]) aggregate values are only stored if they have a certain user specified minimum support. Another possible approach is to introduce parallel processing, which can provide two key ingredients for dealing with the data cube size: increased computational power through multiple processors, and increased I/O bandwidth through multiple parallel disks (e.g. [3, 4, 6–9, 12–14, 21, 22]). In [23], Ng et al. combined both of the above approaches and studied various algorithms for parallel iceberg-cube computation on PC clusters. The algorithm of choice in [23], referred to as PT, applies a *hybrid* approach in that it combines top-down data aggregation with bottom-up data reduction. Another well known hybrid sequential method, which we will discuss in more detail later in the paper, is Star-Cubing [30].

In this paper, we further investigate the use of hybrid approaches for the *parallel* computation of iceberg-cube queries. We present a new hybrid method, called "Pipe 'n Prune" (PnP), for iceberg-cube query computation. Our approach combines top-down data aggregation through piping with bottom-up a priori data reduction. The main difference to previous approaches is the introduction of a novel PnP operator, which uses a piping approach to aggregate data and at the same time performs a priori pruning for subsequent group-by computations. Our approach was motivated by the two phase hybrid parallel method PT [23], which first partitions based on BUC style bottom-up computation, and then uses top-down aggregation for building the startup group-by for each partition. Inspired by Star-Cubing [30], our new PnP operator extends this two phase approach towards a complete merge between data aggregation and a priori pruning. PnP is very different from Star-Cubing [30] in that PnP retains top-down data aggregation through piping and interleaves it with iceberg bottom-up data reduction (pruning). An illustration of our approach is sketched in Fig. 2. An important property of our PnP method is that it is composed mainly of linear data scans, and does not require complex in-memory structures. This allows us to extend PnP to external memory computation of very large iceberg-cube queries with only minimal loss of efficiency. In addition, PnP is well suited for shared-nothing parallelization (where processors do not share any memory and all data is partitioned and distributed over a set of disks). Our new parallel external memory PnP method provides close to linear speedup, particularly on those data sets that are hard to handle for sequential methods. In addition, parallel PnP scales well and provides near linear speedup for larger number of processors, thereby also solving the scalability problem for the parallel iceberg-cube method proposed in [23].

This paper makes the following contributions:

- We present a novel PnP operator and "Pipe 'n Prune" (PnP) algorithm for the computation of iceberg-cube queries. The novelty of our method is that it completely interleaves a top-down piping approach for data aggregation with bottom-up a priori data pruning. A particular strength of PnP is that it is very efficient for *all* three of the following scenarios:
  - Sequential iceberg-cube queries.
  - External memory iceberg-cube queries.
  - Parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.
- We performed an extensive performance analysis of PnP for all of the above scenarios using both synthetic and real data sets. In general PnP performs very well for both dense *and* sparse data sets and scales well, providing linear speedup for larger numbers of processors. In [23], Ng et al. observe that for their parallel iceberg-cube method "the speedup from 8 processors to 16 processors is below expectation", and attribute this scalability problem to scheduling and load balancing issues. Our analysis, based on experiments using both real and synthetic data sets, shows that PnP solves these problems and scales well on up to 32 processors.

  In more detail our analysis of PnP for the above three scenarios showed the following:
  - **Sequential iceberg-cube queries:** As a special case, PnP also provides a new *sequential* hybrid method for the computation of iceberg-cube queries. We present an extensive performance analysis of PnP in comparison with BUC [2] and StarCube [30]. We observe that the sequential performance of PnP is very stable even for large variations of data density and data skew. Sequential PnP typically shows a performance between BUC and StarCube, while BUC and StarCube have ranges of data density and skew where BUC outperforms StarCube or vice versa. For the special case of full cube computation PnP outperforms both BUC and StarCube.

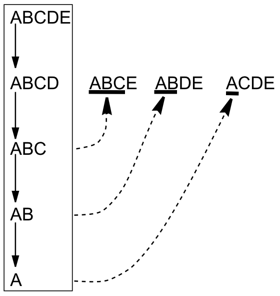
- **External memory iceberg-cube queries:** Since PnP is composed mainly of linear scans, and does not require complex in-memory data structures, it is conceptually easy to implement as an external memory method for very large iceberg-cube queries. In order to make good use of PnP's properties we have implemented our own I/O manager to have full control over latency hiding through overlapping of computation and disk I/O. We present an extensive performance analysis of PnP for external memory computation of very large iceberg-cube queries. Our experiments show minimum loss of efficiency when PnP switches from in-memory to external memory computation. The measured external memory running time (where PnP is forced to use external memory by limiting the available main memory) is only slightly higher than the running time for full in-memory computation of the same iceberg-cube query.
- **Parallel iceberg-cube queries on shared-nothing PC clusters (with multiple disks):** PnP is well suited for shared-nothing parallelization, where processors do not share any memory and all data is partitioned and distributed over a set of disks. We present a PnP parallelization which (1) minimizes communication overhead, (2) balances work load, and (3) makes full use of our I/O manager by overlapping *parallel* computation and *parallel* disk access on all available disks in the PC cluster. Extensive experiments show that our new parallel external memory PnP method provides close to linear speedup particularly on those data sets that are hard to handle for sequential methods. Most importantly, parallel PnP scales well, and provides near linear speedup for larger numbers of processors, thereby also solving an important open scalability problem observed in [23].

The remainder of this paper is organized as follows: Sect. 2 provides first a high level overview of our PnP approach, and then presents the algorithmic details for the three scenarios mentioned above. Section 3 presents an in-depth performance evaluation of PnP, and Sect. 4 concludes our paper.

2 The PnP algorithm

PnP is a hybrid sort-based algorithm for the computation of very large iceberg-cube queries. The idea behind PnP is to fully integrate data aggregation via top-down piping [26] with bottom-up (BUC [2]) a priori pruning. We introduce a new operator, called the *PnP operator*. For a group-by v, the PnP operator performs two steps: (1) It builds all group-bys v' that are a prefix of v through one single sort/scan operation (piping [26]) with iceberg-cube pruning. (2) It uses these prefix group-bys to perform bottom-up (BUC [2]) a priori pruning for new group-bys that are starting points of other piping operations. An example of a 5-dimensional PnP operator is shown in Fig. 1. The PnP operator is applied recursively until all group-bys of the iceberg-cube have been generated. An example of a 5-dimensional *PnP Tree* depicting the entire process for a 5-dimensional iceberg-cube query is shown in Fig. 2. The remainder of this section describes in detail our PnP method for the following three scenarios:

Fig. 1 A PnP operator



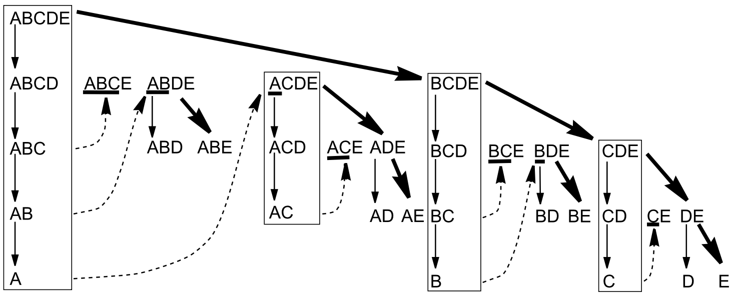


Fig. 2 A PnP tree. (*Plain arrow*: top-down piping. *Dashed arrow*: bottom-up pruning. *Bold arrow*: sorting)

- Sequential, in memory, iceberg-cube queries.
- External memory iceberg-cube queries.
- Parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.

2.1 PnP: sequential in-memory version

We assume as input a table $R[1..n]$ representing a d-dimensional raw data set R consisting of n rows $R[i]$, $i = 1, \dots, n$. Because of the iceberg-cube constraint, a cell in a cuboid is only returned if it has *minimum support*. That is, a cell is only calculated if there are at least *min_sup* tuples assigned to that cell, for some given input parameter *min_sup*.

For a row $R[i]$ we denote with $\underline{R}_j[i]$ the prefix of $R[i]$ consisting of the first j feature values of $R[i]$, followed by the measure value of $R[i]$. We denote with $\hat{R}^j[i]$ the row $R[i]$ with its feature value in dimension j removed. We denote with $\emptyset$ the empty (0-dimensional) group-by. For a group-by v we denote with $|v|$ the number of dimensions of v, and with $\hat{v}^j$ the group-by that is the same as v but with dimension j removed. We denote with $\underline{v}_j$ the group-by identifier consisting of the first j dimensions of v.

Our PnP method for the sequential, in memory, case is shown in Algorithms 1 and 2. Algorithm 2 represents the main part, the implementation of the recursive PnP operator.

Algorithm 1 Algorithm PnP: sequential, in memory

```
Input: $R[1..n]$: a table representing a $d$-dimensional raw data set consisting of $n$
    rows $R[i]$, $i = 1, \dots, n$; $min\_sup$: the minimum support.
Output: The iceberg data cube.
1: Sort $R$ and aggregate duplicates in $R$.
2: Call PnP-1($R$, $v_R$, $\emptyset$), where $v_R$ is the group-by containing all dimensions of $R$
   (sorted by cardinality in decreasing order).
```

We explain our algorithm using the example in Fig. 2 for a 5-dimensional iceberg-cube query. In Line 2 of Algorithm 1, we call PnP-1(R, ABCDE, ∅). This will first result in the creation of the pipe ABCDE–ABCD–ABC–AB–A and then create pruned versions of ABCE, ABDE, and ACDE for subsequent piping operations. Table 1 shows a complete execution for the example raw data set R indicated in the first column of Table 1. Buffers $b[5] \cdots b[1]$ represent the results of piping operations, while $R_3 \cdots R_1$ show the result of pruning operations. Note that the PnP operator uses only one single pass through the data set. The horizontal lines in Table 1 indicate cases where aggregation or pruning take place. The recursive call in Line 12 of Algorithm 2 initiates the PnP operator for group-bys ABCE, ABDE, and ACDE. The prefix passed as third parameter in Line 12 of Algorithm 2 is shown in Fig. 2 as the underlined portions of ABCE, ABDE, and ACDE, respectively. It represents for those recursive calls the portion of the pipe that has already been computed. The recursive call in Line 20 of Algorithm 2 initiates the PnP operator for group-by BCDE and starts the iceberg-cube computation for all group-bys not containing A. The resulting entire process is depicted in Fig. 2.

2.2 PnP: sequential external memory version

Since PnP is sort based, it is easy to extend PnP to external memory, as shown in Algorithms 3 and 4. We discuss here only the main differences between Algorithm 2 and Algorithm 4. All sort operations are replaced by external memory sorts. Some care has to be taken with the scan and aggregation/pruning operations, as buffers may overflow and have to be saved to disk. The main difference between Algorithm 2 and Algorithm 4 is with respect to the recursive calls in Line 12 in Algorithm 2. In the external memory version, we have to save the tables R_j into a file F_j on disk as shown in Line 11 of Algorithm 4. A separate loop in Lines 18 to 22 of Algorithm 4 is then required to retrieve all R_j and perform the recursive calls. Note that these operations are independent and we can apply disk latency hiding through overlapping of computation and disk I/O. In order to make good use of this effect, we have implemented our own I/O manager which resulted in a significant performance improvement (see Sect. 3.2).

2.3 PnP: parallel and external memory version

We now discuss how our PnP algorithm can be parallelized in order to be executed on a shared-nothing multiprocessor as shown in Fig. 3a. Such a multiprocessor consists

Algorithm 2 PnP-1(R, v, pv)

```
Input: $R[1..n]$: a table representing the raw data set consisting of $n$ rows $R[i]$, $i = 1, \dots, n$; $v$: identifier for a group-by of $R$; $pv$: a prefix of $v$.
Output: The iceberg data cube.
1: **Local Variables:** $k = |v| - |pv|$; $R_j$: tables for storing rows of $R$; $b[1..k]$: a buffer for storing $k$ rows, one for each group-by $\underline{v}_1 \cdots \underline{v}_k$; $h[1..k]$: $k$ integers; $i$, $j$: integer counters. **Initialization:** $b[1..k] = [\text{null..null}]$; $h[1..k] = [1..1]$.
2: for $i = 1..n$ do
3:   for $j = k..1$ do
4:     if ($b[j] = \text{null}$) OR (the feature values of $b[j]$ are a prefix of $R[i]$) then
5:       Aggregate $\underline{R}_j[i]$ into $b[j]$.
6:     else
7:       if $b[j]$ has minimum support then
8:         Output $b[j]$ into group-by $\underline{v}_j$.
9:         if $j \le k - 2$ then
10:          Create a table $R_j = \hat{R}^{j+1}[h[j]] \cdots \hat{R}^{j+1}[i-1]$.
11:          Sort and aggregate $R_j$.
12:          Call **PnP-1**($R_j$, $\hat{v}^{j+1}$, $\underline{v}_j$).
13:        end if
14:      end if
15:      Set $b[j] = null$ and $h[j] = i$.
16:    end if
17:  end for
18: end for
19: Create a table $R'[1..n']$ by sorting and aggregating $\hat{R}^1[1] \cdots \hat{R}^1[n]$.
20: Call **PnP-1**($R'$, $\hat{v}^1$, $\emptyset$).
```

Algorithm 3 Algorithm PnP: sequential, external memory

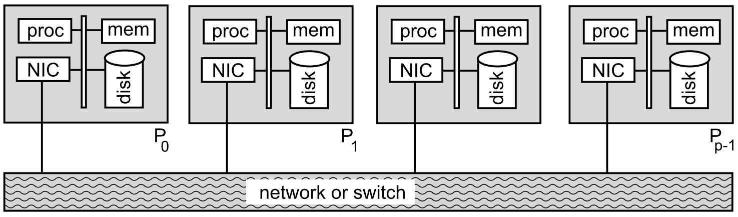
```
Input: $R[1..n]$: a table (stored on disk) representing a $d$-dimensional raw data set consisting of $n$ rows $R[i]$, $i = 1, \dots, n$; $min\_sup$: the minimum support.
Output: The iceberg data cube (stored on disk).
1: Sort $R$, using external memory sorting, and aggregate duplicates in $R$.
2: Call **PnP-2**($R$, $v_R$, $\emptyset$), where $v_R$ is the group-by containing all dimensions of $R$ (sorted by cardinality in decreasing order).
```

of p processors $P_0 \cdots P_{p-1}$, each with its own memory and disk. The processors are connected via a network or switch. Our focus is on practical parallel methods that can be implemented on low-cost PC clusters consisting of standard Intel processor based Linux machines connected via Gigabit Ethernet. However, our methods can also be used, and will perform even better, on more expensive platforms such as clusters connected via Myrinet or shared memory parallel machines like the SunFire.

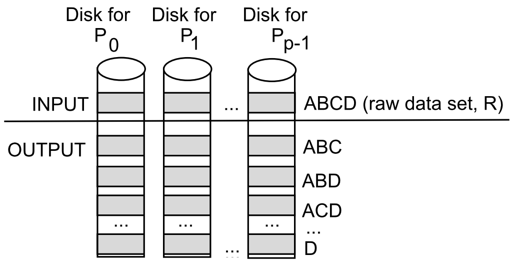
We assume as input a d-dimensional raw data set R stored in a table consisting of n rows that are distributed over the p processors as shown in Fig. 3b. More precisely,

Table 1 PnP processing of $ABCDE$

R	$b[5]$	$b[4]$	$b[3]$	R_3	$b[2]$	R_2	$b[1]$	R_1
ABCDE	**ABCDE**	**ABCD**	**ABC**	**ABCE**	**AB**	**ABDE**	**A**	**ACDE**
11111 1	11111 1			pruned		1111 2		1111 1
11112 1	11112 1	1111 2				1112 1		1112 1
11122 1	11122 1	1112 1	111 3			1122 1		1122 1
11211 1	11211 1	1121 1	112 1	pruned	**11 4**		**1 4**	1211 1
21111 1	21111 1	2111 1		pruned		pruned		pruned
21121 1	21121 1							
21122 1	21122 1	2112 2	211 3		21 3		2 3	
31111 1	31111 1	3111 1		pruned		3111 1		3111 1
31121 1	31121 1					3121 2		3121 1
31122 1	31122 1	3112 2	311 3			3122 1		3122 1
31221 1	31221 1			pruned		3123 1		3221 1
31223 1	31223 1	3122 2	312 2		**31 5**		**3 5**	3223 1
41111 1	41111 1	4111 1	411 1	pruned	41 1	pruned		4111 2
42111 1	42111 1			pruned		pruned		4112 1
42112 1	42112 1	4211 2	421 2		42 2			4121 1
43121 1	43121 1	4312 1	431 1	pruned	43 1	pruned	**4 4**	



(a)



(b)

Fig. 3 Computing platform: (**a**) shared-nothing multiprocessor (**b**) with individual views stripped across the distributed disks

every processor P_i stores on its disk a table R_i consisting of $\frac{n}{p}$ rows of R. As indicated in Fig. 3b, each group-by of the output (iceberg-cube) will also be partitioned and distributed over the p processors. We refer to this process as *striping* a table over the

Algorithm 4 PnP-2(R, v, pv)

```
Input: $R[1..n]$: a table (stored on disk) representing the raw data set consisting of $n$
    rows $R[i]$, $i = 1, \ldots, n$; $v$: identifier for a group-by of $R$; $pv$: a prefix of $v$.
Output: The iceberg data cube (stored on disk).
 1: Local Variables: $k = |v| - |pv|$; $R_j$: tables for storing rows of $R$ (called partitions); $F_j$: disk files for storing multiple partitions $R_j$; $b[1..k]$: a buffer for storing $k$ rows, one for each group-by $\underline{v}_1, \ldots, \underline{v}_k$; $h[1..k]$: $k$ integers; $i$, $j$: integer counters. Initialization: $b[1..k] = [\text{null..null}]$; $h[1..k] = [1..1]$.
 2: for $i = 1..n$ (while reading $R[i]$ from disk in streaming mode...) do
 3:   for $j = k..1$ do
 4:     if ($b[j] = null$) OR (the feature values of $b[j]$ are a prefix of $R[i]$) then
 5:       Aggregate $\underline{R}_j[i]$ into $b[j]$.
 6:     else
 7:       if $b[j]$ has minimum support then
 8:         Output $b[j]$ into group-by $\underline{v}_j$. Flush to disk if $\underline{v}_j$'s buffer is full.
 9:         if $j \le k - 2$ then
10:           Create a table $R_j = \hat{R}^{j+1}[h[j]] \cdots \hat{R}^{j+1}[i-1]$.
11:           Sort and aggregate $R_j$ (using external memory sort if necessary).
              Write the resulting $R_j$ and an "end-of-partition" symbol to file $F_j$.
12:         end if
13:       end if
14:       Set $b[j] = null$ and $h[j] = i$.
15:     end if
16:   end for
17: end for
18: for $j = k..1$ do
19:   for each partition $R_j$ written to disk file $F_j$ in line 11 do
20:     Call PnP-2($R_j$, $\hat{v}^{j+1}$, $\underline{v}_j$).
21:   end for
22: end for
23: Create a table $R'[1..n']$ by sorting and aggregating $\hat{R}^1[1] \cdots \hat{R}^1[n]$ (using external memory sort if necessary).
24: Call PnP-2($R'$, $\hat{v}^1$, $\emptyset$).
```

p disks. When every group-by is striped over the p disks, access to a group-by can be performed with maximum I/O bandwidth through full parallel disk access.

For a d-dimensional table R_i, we define tables T_i^j, $j = 1, \ldots, d$, as the tables obtained by removing from each row of R_i the first $j - 1$ feature values and performing aggregation to remove duplicates (but not performing iceberg-cube pruning). Note that, $T_i^1 = R_i$.

Our parallel PnP method is shown in Algorithm 5. The basic idea is illustrated in Fig. 4. The figure shows a *PnP forest* obtained by partitioning the PnP tree in Fig. 2 into d trees, one for each feature dimension. The data set for the root of the j-th tree is the set $T^j = T_0^j \cup T_1^j \cup \cdots \cup T_{p-1}^j$. We start with $T^1 = R$ striped over the p disks, where processor P_i stores $T_i^1 = R_i$, and execute on each processor P_i

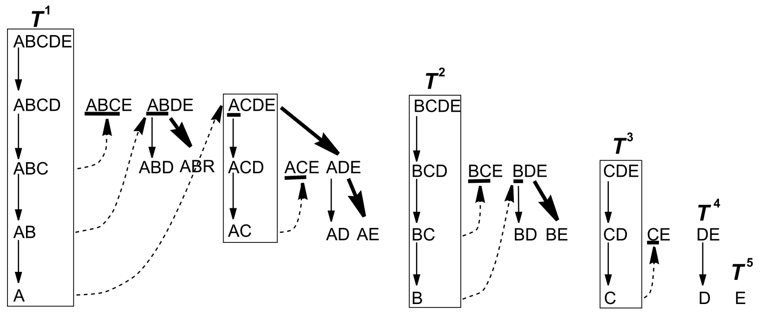


Fig. 4 A PnP forest

Algorithm 5 Algorithm PnP: parallel, external memory

```
Input: $R$: a table representing a $d$-dimensional raw data set consisting of $n$ rows,
    stored on $p$ processors. Every processor $P_i$ stores (on disk) a table $R_i$ of $\frac{n}{p}$ rows
    of $R$ as shown in Fig. 3b. $min\_sup$: the minimum support.
Output: The iceberg data cube (distributed over the disks of the $p$ processors as
    shown in Fig. 3b).
1: Variables: On each processor $P_i$ a set of $d$ tables $T_i^1, \ldots, T_i^d$.
2: for $j = 1..d$ do
3:     Each processor $P_i$: Compute $T_i^j$ from $T_i^{j-1}$ via sequential sort. ($T_i^1 = R_i$)
4:     Perform a parallel global sort on $T_1^j \cup T_1^j \cup \cdots \cup T_p^j$.
5: end for
6: for $j = 1..d$ do
7:     Each processor $P_i$: Apply Algorithm 3 to $T_i^j$.
8: end for
```

the sequential Algorithm 3 with input T_i^1 (Line 7 of Algorithm 5). This creates the first tree in the PnP forest of Fig. 4. Next, we compute on each processor P_i the table T_i^2 from T_i^1 by removing the first feature dimension and performing aggregation to remove duplicates (via a sequential sort); see Line 3 of Algorithm 5. Different data aggregation on different processors can lead to imbalance between processors, and the set $T_0^2 \cup T_1^2 \cup \cdots \cup T_{p-1}^2$ is therefore re-balanced through a global sort (Line 4 of Algorithm 5). We can then execute on each processor P_i the sequential Algorithm 3 with input T_i^2 (Line 7 of Algorithm 5), creating the second tree in the PnP forest of Fig. 4. This process is iterated d times, until all group-bys have been built.

3 Performance evaluation

We have implemented sequential, external memory, and parallel versions of our PnP algorithm as presented in the previous section. The sequential C++ code evolved from the code for top-down sequential pipesort used in [4]. The external memory code evolved from the sequential code through the addition of an external memory sort and a custom designed I/O manager that allows us to overlap computation and disk I/O. The parallel code was built from our external memory code base together with communication operations drawn from the MPI communication library.

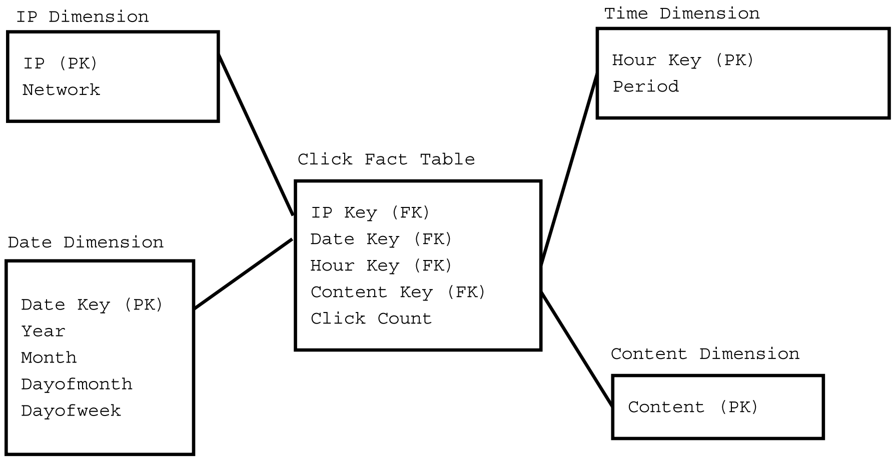
Our performance evaluation was conducted in two stages. In the first stage we evaluate the sequential version of PnP by comparing it with implementations of BUC and Star-Cubing. The Microsoft Windows executables for these implementations were kindly provided by J. Han's research group to enable just such comparative performance testing of cube construction methods [16]. The PnP codes, for both the sequential (in-memory) version and external memory version, were compiled using Visual C++ 6.0. Both sequential and external memory experiments were conducted on a 2.8 GHz Intel Pentium 4 based PC running Microsoft Windows 2000 with 1 GB RAM and an 80 GB 7200 RPM IDE disk.

In the second stage of our performance evaluation we explored the performance of our parallel version of PnP on a 32 processor cluster. This shared nothing parallel machine consists of a collection of 1.7 GHz Intel Xeon processors each with 1 GB of RAM, two 40 GB 7200 RPM IDE disks and an onboard Inter Pro 1000 XT NIC. Each processor is running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6. as part of a ROCKS cluster distribution. All processors are interconnected via a Cisco 6509 GigE switch. Due to restrictions in machine access, we were frequently unable to reserve all 32 processors of this machine. In such cases a minimum of 16 processors were used.

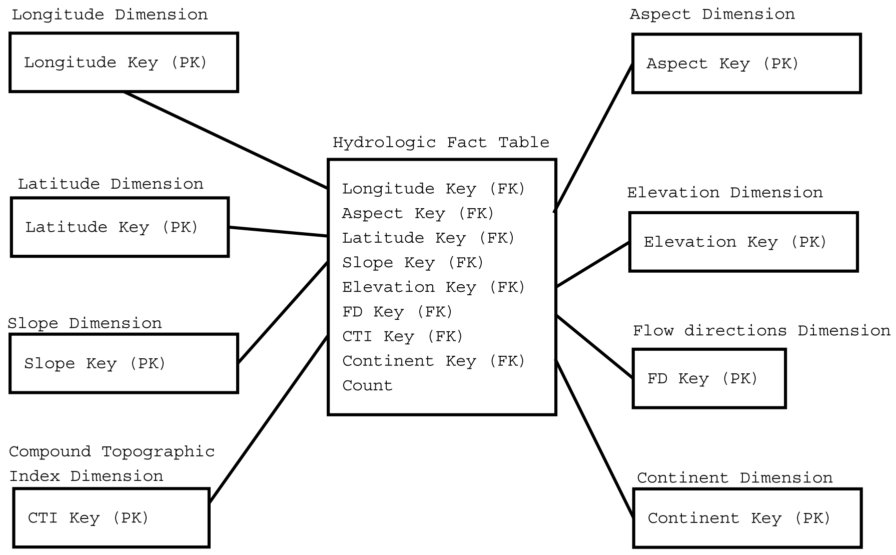
In the following experiments, all sequential times are measured as wall clock times in seconds. All parallel times are measured as the wall clock time between the start of the first process and the termination of the last process. All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times are measured with no other users on the machine. The running times for BUC and Star-Cubing that we show were observed by running the executables obtained from [16].

To fully explore the performance of these cube construction methods, we generated a large number of synthetic data sets which varied in terms of the following parameters: n—the number of rows in the raw data set R, d—the number of dimensions, s—the skew in each dimension as a zifp value, m—the minimum support, b—the available memory in bytes, and $c_1, \dots, c_d$—the cardinalities of the d dimensions (where an unsubscripted c indicates the same cardinality in all dimensions). The data generator used in the sequential and external memory experiments to generate the raw data set R was provided with the BUC and Star-Cubing executables from [16]. For the parallel experiments we generated similar synthetic data sets using our own data generator which creates striped data across the p disks and which had been previously used in [3, 4].

We also evaluated the performance of PnP on two large *real* data sets each consisting of over a 100 M rows of data. The first data set was constructed from 12 months of web log files from a local newspaper website using ETL processing. In the original log files each line indicates a mouse click event and consists of the IP address, date, time and page accessed. The log files were transformed into the star-schema shown in Fig. 5 where the base table consists of four dimensions (IP, date, time and content)



(a)



(b)

Fig. 5 Star schema for (**a**) the News Web Log data set, and (**b**) the World Hydrological data set

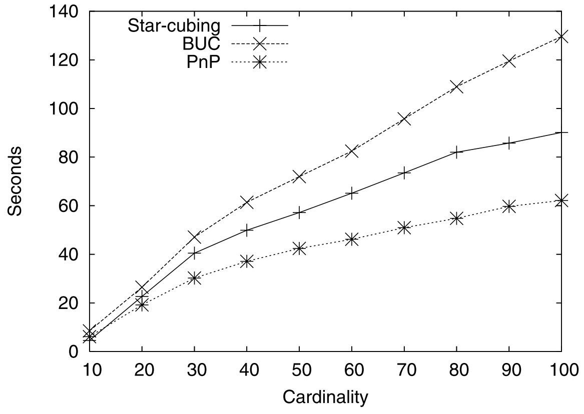
and one measure (mouse click counts). The IP dimension includes 833471 unique addresses, the date dimension 366 unique days, the time dimension 24 unique hours, and the content dimension 25 unique categories of page contents. In total, the main fact table consists of over 32 M rows.

The second data set comes from the *HYDRO1k Elevation Derivative Database* [18]. This geographic database provides hydrologic information on a continental scale. The database includes the data for six continents: North America, South America, Europe, Africa, Asia and Australia. For each continent, it divides the surface area into equal size squares based on longitude and latitude and for each square it records a range of geographic information. In the main table, there are eight dimensions: longitude, latitude, aspect, slope, elevation, flow directions, compound topographic index and continents with cardinalities of 360, 360, 180, 90, 82, 8, 7 and 6 respectively. The table includes one measure: square counts. In total, the main fact table consists of 17 M rows.

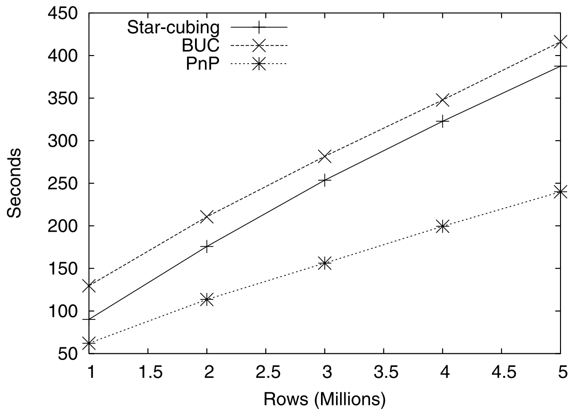
3.1 Sequential experiments

The performance results for our sequential experiments are shown in Figs. 6 to 12. There are three groups of experiments in this section. The first group, Fig. 6, compares PnP to BUC and Star-Cubing for the special case of full cube computation. The second group, Figs. 7 to 8 compares the iceberg cube computation on raw data sets of varying sparsity. The last group, Figs. 9 to 11 explores various settings of the size of raw data, dimensions, skew and the minimum support for both very dense and very sparse cubes.

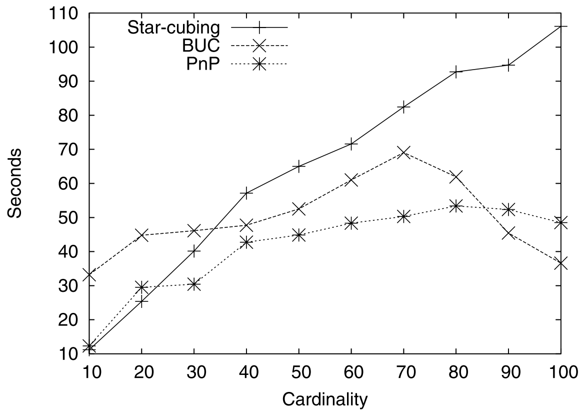
Fig. 6 Sequential PnP. (**a**) Full cube running time in seconds as a function of cardinality ($n = 1$ M) and (**b**) full cube running time in seconds as a function of the size of raw data set ($|D_i| = 256$, $1 \le i \le d$). (Fixed parameters: $d = 6$)



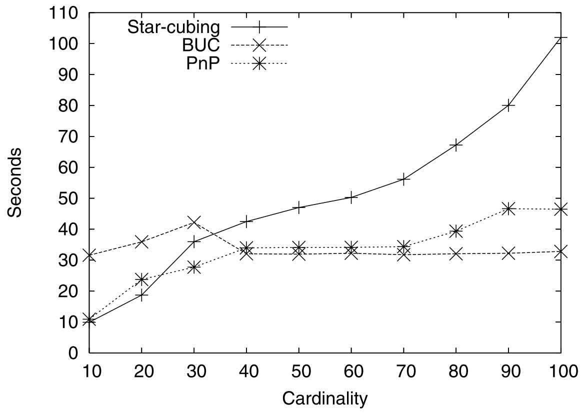
(a)



(b)



(a)



(b)

Fig. 7 Sequential PnP. Running time in seconds as a function of cardinality (**a**) $m = 10$ and (**b**) $m = 100$. (Fixed parameters: $n = 5$ M, $d = 6$)

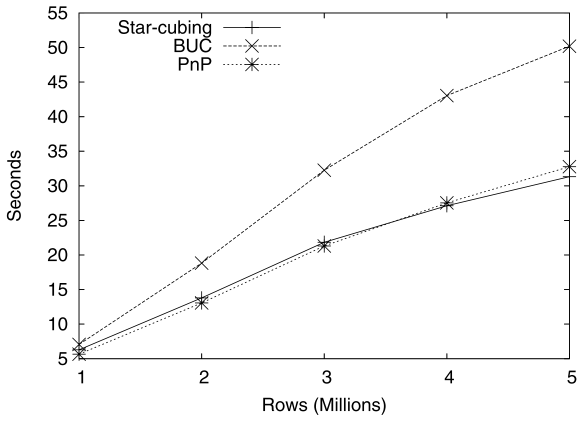
Figure 6 shows for full cube computation (i.e., minimum support $m = 1$) results for PnP compared to BUC and Star-Cubing for various cardinalities and data sizes. Note that varying cardinality, while holding the other parameters constant, amounts to varying the sparsity. We observe that for the special case of full cube computation the sequential version of PnP performs better than BUC or Star-Cubing regardless of sparsity. In this case, PnP takes full advantage of pipeline processing and saves significant time by sharing sorts, while bottom-up a priori data pruning is ineffectual.

Figure 7 compares PnP to BUC and Star-Cubing for iceberg cube computation with varying sparsity. In order to measure the sparsity of a data cube, we use the following formula for its calculation:

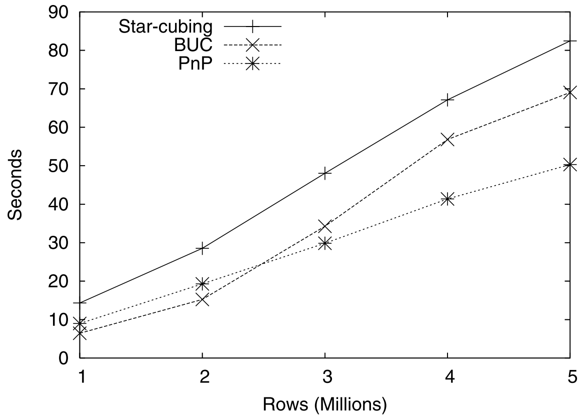
$$Sp = \frac{n}{\prod_{i=1}^{d} |D_i|}.$$

In Fig. 7a, sequential PnP typically shows the best performance when the cardinality is between 25 and 80, which corresponds to a data cube sparsity Sp between 0.02

Fig. 8 Sequential PnP. Running time in seconds as a function of the size of raw data set (**a**) $|D_i| = 22$ and (**b**) $|D_i| = 80$. (Fixed parameters: $d = 6$, $m = 10$)



(a)

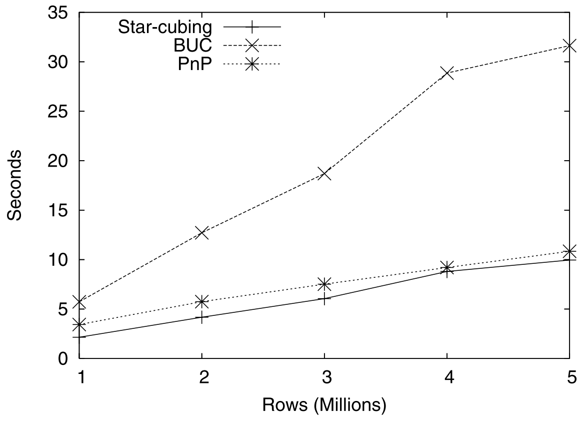


(b)

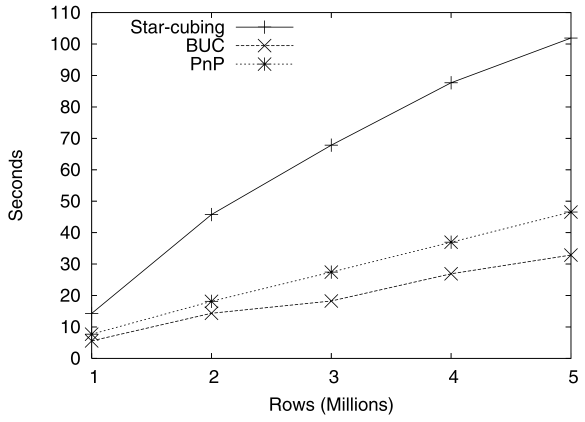
and 0.00002. Star-Cubing is the best method when the cardinality is below 25, or *Sp* larger than 0.02, which is a very dense case. BUC is the best one when the cardinality is above 80, or *Sp* larger than 0.00002, which is a very sparse case. Figure 7b shows the results when the minimum support is 100. We observe a trend similar to Fig. 7a. PnP and BUC are very similar when *Sp* is between 0.02 and 0.00002, and both of them are better than Star-Cubing. When *Sp* is larger than 0.02, Star-Cubing is best, and when *Sp* is less than 0.00002, BUC is the best.

In order to take a close look at the performance for the threshold values 0.02 and 0.00002, we selected cardinalities of 22 and 80, respectively, for a range of values for n, in order to examine at which point PnP switches position with BUC or Star-Cubing. Figure 8a shows the running time for varying data size with the cardinality for each dimensions set to $|D_i| = 22$. In this dense cube case, we observe that PnP and Star-cubing are better than BUC, and they switch position between 3 M and 4 M. The sparsity at 3 M is about 0.02. When the data size is smaller than 3 M, or the sparsity smaller than 0.02, PnP is better than Star-cubing. When the sparsity is increased and

Fig. 9 Sequential PnP. Running time in seconds as a function of n the size of raw data (**a**) $|D_i| = 10$ and (**b**) $|D_i| = 100$. (Fixed parameters: $d = 6$, $m = 10$)



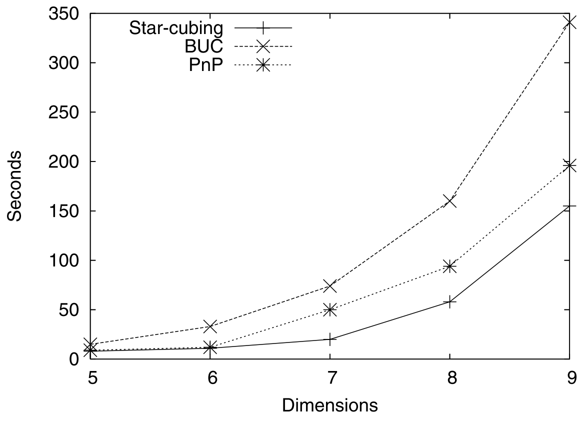
(a)



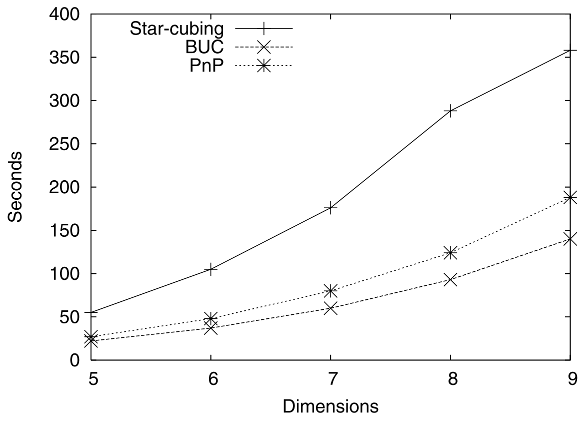
(b)

larger than 0.02, Star-cubing becomes better than PnP. Figure 8b shows the running time for varying data size with the cardinality for each dimensions set to $|D_i| = 80$. This is a sparse cube case. We observe that PnP and BUC are better than Star-cubing, and they switch position between 2 M and 3 M. The sparsity at 2 M is about 0.00002. When the data size is smaller than 2 M, or the sparsity smaller than 0.00002, BUC is better than PnP. When the sparsity is larger than 0.00002, PnP becomes better than BUC.

Finally, Figs. 9 to 12 compare PnP to BUC and Star-Cubing for iceberg cube computation while varying the raw data size, dimensionality, minimum support, and skew, for both very dense ($|D_i| = 10$ or $Sp = 5$) and very sparse ($|D_i| = 100$ or $Sp = 0.000005$) cubes. We observe that the sequential performance of PnP is very stable even in these extreme cases. Sequential PnP performance is almost always close to the best one, while BUC tends to perform best on very sparse data sets and Star-Cubing best on very dense data sets.



(a)



(b)

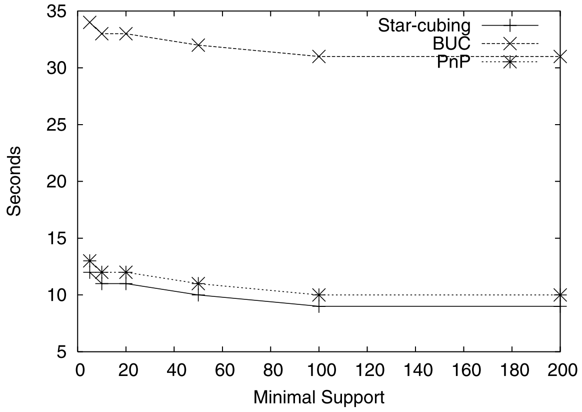
Fig. 10 Sequential PnP. Running time in seconds as a function of n the size of raw data set (**a**) $|D_i| = 10$ and (**b**) $|D_i| = 100$. (Fixed parameters: $d = 6$, $m = 10$)

Overall, sequential PnP shows the best performance when the data sets are in the normal range of sparsity ($0.00002 \le Sp \le 0.02$), and it still performs close to the best of BUC and Star-Cubing in the extremely sparse or dense cases. Therefore sequential PnP appears to be an interesting alternative to BUC and Star-Cubing especially in applications where performance stability over a large wide range of input parameters is important.

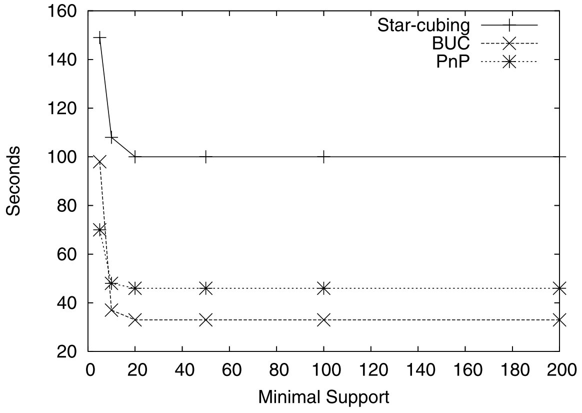
3.2 External memory experiments

Our performance results for the external memory version of PnP are shown in Fig. 13. Note that since PnP is composed mainly of linear scans and does not require complex in-memory data structures, it is reasonably easy to implement as an external memory method for very large iceberg-cube queries. In order to make good use of PnP's properties, we implemented our own I/O manager in order to have full control over disk access patterns as well as latency hiding through overlapping of computation and disk I/O. For the evaluation of the external memory version of PnP we

Fig. 11 Sequential PnP. Running time in seconds as a function of minimum support (**a**) $|D_i| = 10$ and (**b**) $|D_i| = 100$. (Fixed parameters: $n = 5$ M. $d = 6$)



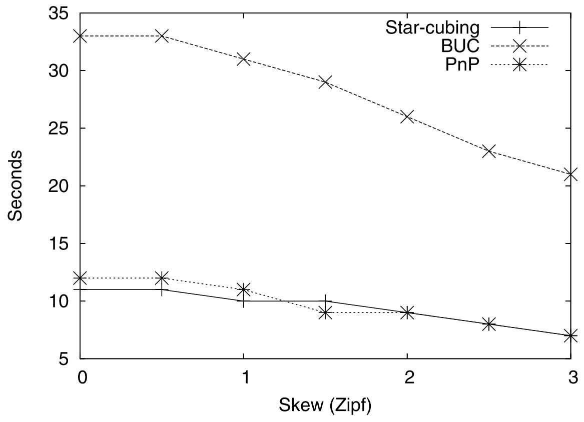
(a)



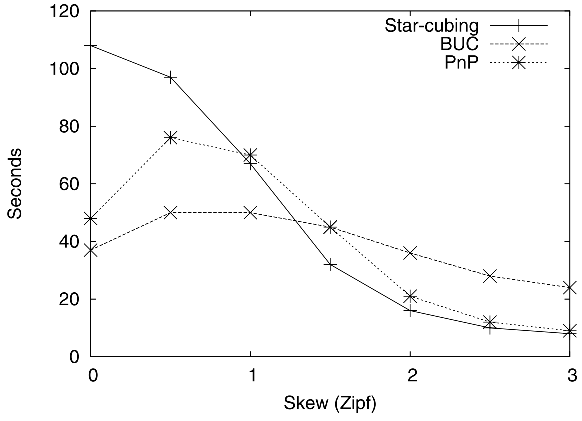
(b)

used larger data sets, ranging in size from 1 million to 20 million rows, while varying dimensionality d and available memory M.

Overall, our experiments show minimum loss of efficiency when PnP switches from in-memory to external memory computation. The measured external memory running time (where PnP is forced to use external memory by limiting the available main memory) is only slightly higher than the running time for full in-memory computation of the same iceberg-cube query. In Fig. 13a we observe similarly shaped curves even as we increase the dimensionality of the problem, due in large part to the effects of iceberg pruning. The location of the slight jump in time, corresponding to the switch to external memory, occurs between 5 million rows and 7 million rows depending on the dimensionality of the iceberg cube being generated. Figure 13b shows, not surprisingly, that there is a benefit to increasing the memory space M available to the external memory algorithm. However, the gaps between the curves are very small. It suggests that the external memory PnP algorithm makes full use of available memory and exhibits good performance even with limited memory footprints.



(a)



(b)

Fig. 12 Sequential PnP. Running time in seconds as a function of the data skew (**a**) $|D_i| = 10$ and (**b**) $|D_i| = 100$. (Fixed parameters: $n = 5$ M, $d = 6$)

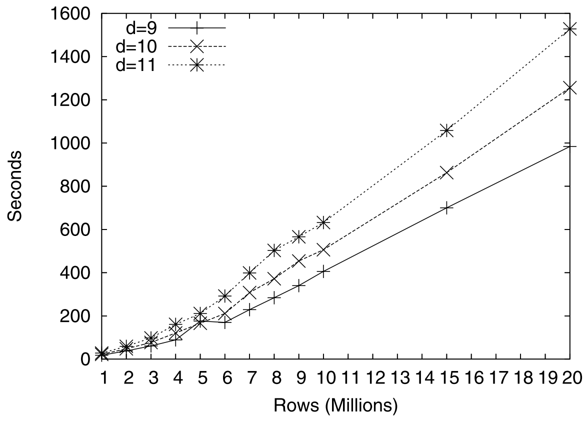
3.3 Parallel experiments on synthetic data sets

Our performance results for the parallel shared-nothing version of PnP are shown in Figs. 14 to 17. This implementation is based on the external memory PnP code base with MPI code added for the parallel part. In addition to parallelism, our code uses external memory processing when needed.

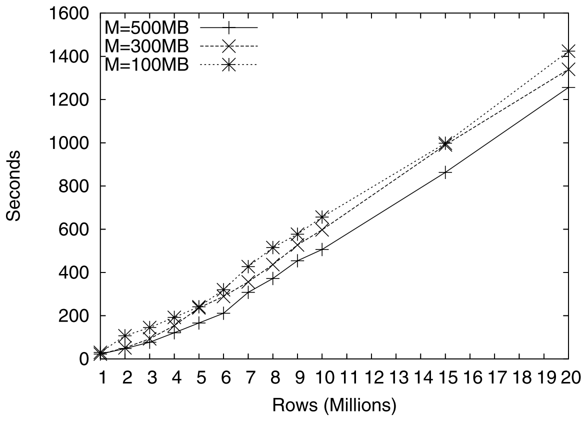
Our experiments focus on speedup, since this is one of the key metrics for the evaluation of parallel database systems [10]. The experiments consist of increasing the number of processors available to the parallel version of PnP to determine the time and corresponding parallel speedup obtained while varying other key parameters such as input data size, dimensionality, cardinality, minimum support, and skew.

Figure 14a shows the running time of parallel PnP for input data sizes between 1 and 8 million rows and Fig. 14b shows the corresponding speedup. As is typically the case, relative speedup improves as we increase the size of the input and

Fig. 13 External memory PnP. Running time in seconds as a function of the size of raw data set (**a**) $M = 500$ MB, and (**b**) $d = 10$. (Fixed parameters: $|D_i| = 300$, $1 \leq i \leq d$, $m = 100$)



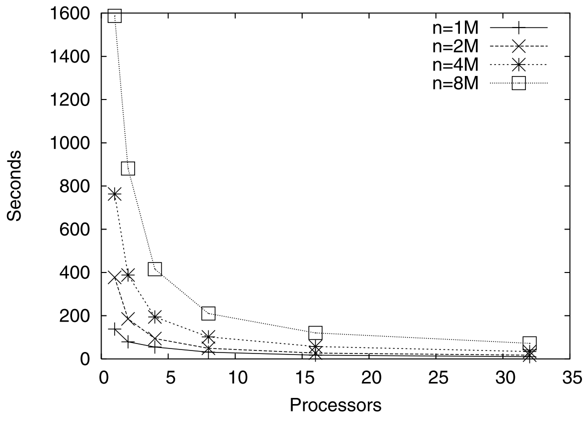
(a)



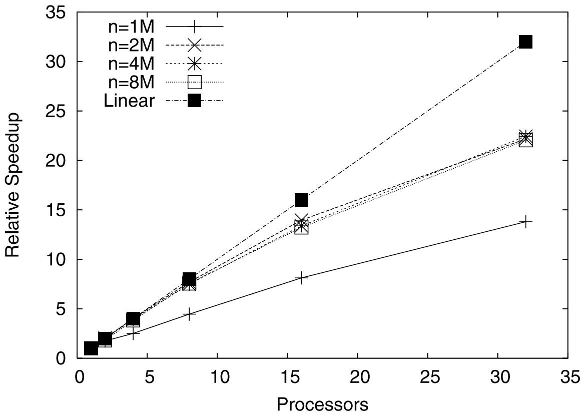
(b)

consequentially the total amount of work to be performed. With data between 2 M and 8 M rows, near optimal linear speedup is observed up to 8 processors. For 16 and 32 processors, speedup drops off somewhat because the problem allocated to each processor is too small to dominate the communication overhead. However, even with only 1 M rows, speedup is optimal up to 4 processors and runs at about 50% for 8, 16 or 32 processors, showing that speedup is achievable even for small data sets.

Figure 15a shows the running time of the parallel version of PnP for increasing dimensionality and Fig. 15b shows the corresponding speedup. We observe that the speedup increases as the number of dimensions increases. Note that when the number of dimensions increases there are exponentially more views in the data cube, implying a large increase in local computation time. Communication time also increases but much more slowly. In fact, for twice the number of views, PnP requires just one additional parallel sort. The growth in local computation time clearly dom-



(a)



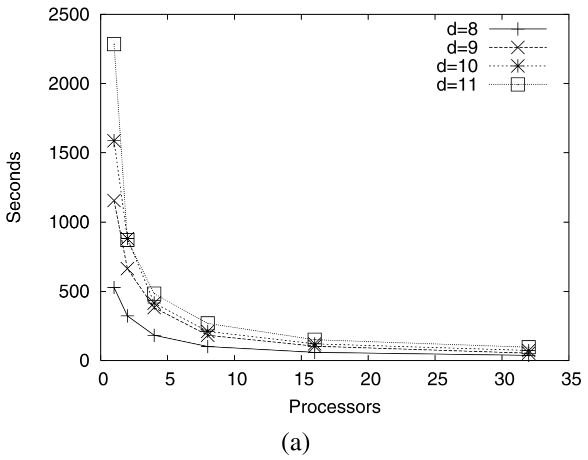
(b)

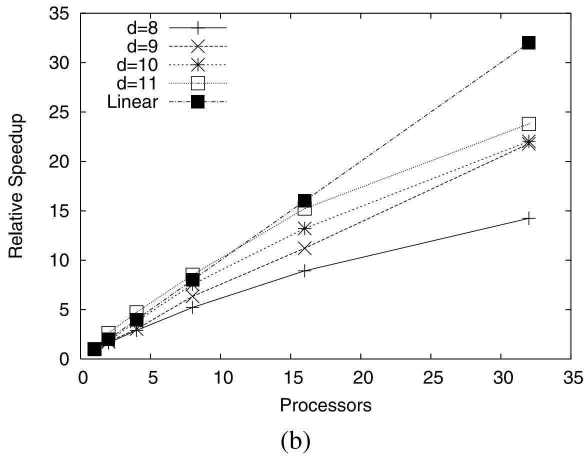
Fig. 14 Parallel external memory PnP. (**a**) Parallel wall clock time in seconds as a function of the number of processors for data size n from 1 million to 8 million rows and (**b**) corresponding speedup. (Fixed parameters: $d = 10$, $|D_i| = 100$, $1 \le i \le d$, $m = 100$, $M = 100$ Mb)

inates the growth in communication overhead leading to improved speedup. Note that, the best speedup is achieved on cubes that are hardest to compute sequentially, i.e., those that involve the largest computation in terms of input size and/or dimensionality.

The cardinality of the dimensions in the input data can significantly affect performance in two ways: (1) the product of the cardinalities affects sparsity, and (2) the ratio of the size of the largest cardinalities to p affects the quality of the parallel load balance. Figure 16a shows the running time of the parallel version of PnP for input data covering a range of cardinalities and Fig. 16b shows the corresponding speedup. Data set D, with cardinalities $|D_i| = 200$, $1 \le i \le d$, is both dense and lacks any large cardinality dimensions. These factors combine to reduce the achieved speedup. Data sets A and C are both more sparse and have larger maximum cardinalities which leads to a better load balance and the improved speedup we observe. For data set A, with the cardinalities $|D_i| = 200$, $1 \le i \le d$, we observe for the first time super linear speedup. In this case we have both large output size and an excellent

Fig. 15 Parallel external memory PnP. (**a**) Parallel wall clock time in seconds as a function of the number of processors for dimensions d from 8 to 11 and (**b**) corresponding speedup. (Fixed parameters: $n = 8$ M, $|D_i| = 100$, $1 \leq i \leq d$, $m = 100$, $M = 100$ Mb)





load balance. The super linearity is due to the fact that as p grows the data allocated to each processor shrinks allowing our parallel external memory PnP algorithm to adaptively switch more of the computation to the more efficient in-memory version of PnP.

Figure 17 shows the effects on running time and speedup of varying minimum support. We observe that for smaller values of minimum support, $m = 100$ and $m = 500$, the computing time required is larger and the speedup obtained by our parallel PnP algorithm is near linear on up to 16 processors and then drops off more sharply towards 32 processors. The curves for $m = 100$ and $m = 500$ are almost overlapped because most measures in the data cube are greater than 500, and therefore also greater than 100, so that the two cases output almost the same amount of data. For the large minimum support cases, $m = 1000$ and $m = 2500$, the speedup drops off faster because much of the data is pruned and there is less local computing to mask the communication overheads.

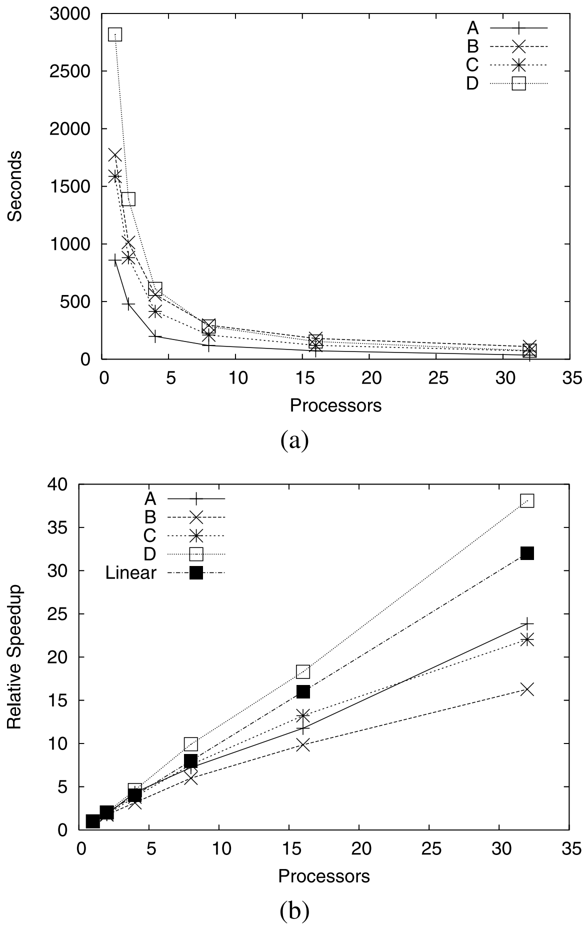
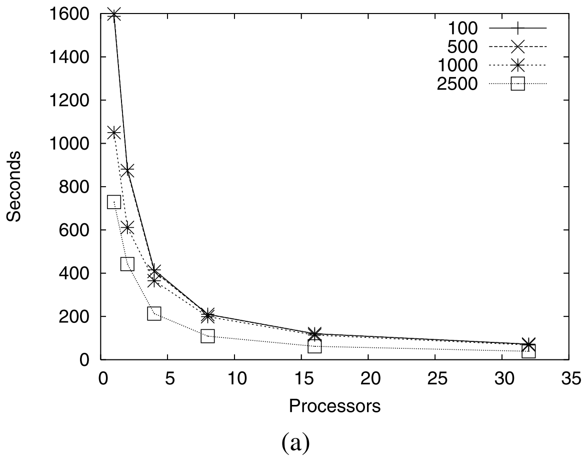


Fig. 16 Parallel external memory PnP. (**a**) Parallel wall clock time in seconds as a function of the number of processors for data sets with different cardinality mixes, and (**b**) corresponding relative speedup. (Fixed parameters: (A) $|D_i| = 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2$. (B) $|D_i| = 50$, $1 \le i \le d$. (C) $|D_i| = 100$, $1 \le i \le d$. (D) $|D_i| = 200$, $1 \le i \le d$, $n = 8$ M, $d = 10$, $m = 100$, $M = 100$ Mb)

3.4 Parallel experiments on real data sets

In this section the performance characteristics of the PnP algorithm is explored using two large "real" data sets, namely the News Web Log data set and the World Hydrologic data set. One significant difference with these datasets, in comparison to the synthetic ones, is that they contain complex skew, the presence of which may either help or hinder a particular cubing algorithm.

Figure 18 shows the running time and the relative speedup for generating iceberg data cubes from the News Web Log data set. We observe that the speedup is very close to linear in Fig. 18b. In the News Web Log data set, the first and second dimension have large cardinalities compared with the number of processors and, hence, the work load is well balanced by PnP across the processors leading to excellent speedup. Note that, while it is hard to formally describe the amount of skew in this real world data set, we do know that significant skew exists. For example, some IP addresses belong to proxy servers, which have much higher access frequency than personal IP addresses. In addition, many users show a strong preference for one or two content



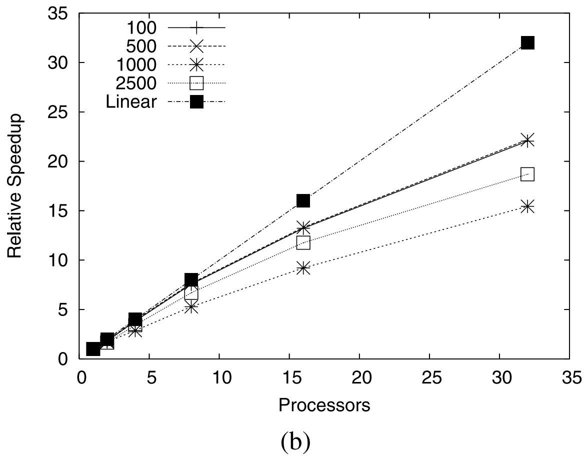
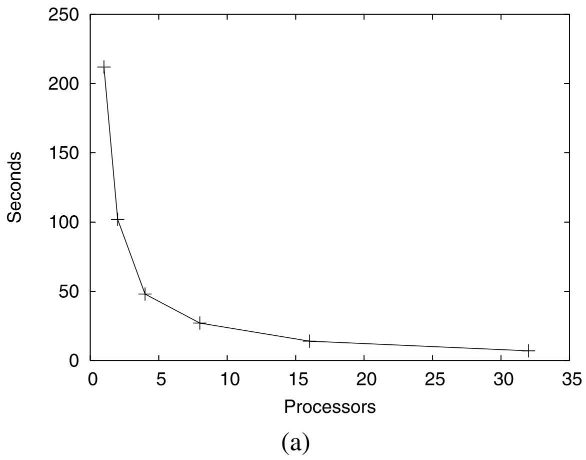


Fig. 17 Parallel external memory PnP. (**a**) Parallel wall clock time in seconds as a function of the minimum support and (**b**) corresponding relative speedup. (Fixed parameters: $n = 8$ M, $d = 10$, $|D_i| = 100$, $1 \le i \le d$, $M = 100$ Mb)

types over all others. In fact about 40% of mouse clicks recorded in the data are on the obituary and birth pages on this local newspaper's web site.[1] In spite of this significant skew, or indeed perhaps because of it, parallel PnP exhibits near to linear speedup.

Figure 19 shows the running time and the relative speedup for generating iceberg data cubes with different minimum support values from the 32 M row World Hydrologic data set. We observe that when the minimum support value is set to 500 (i.e., there is significant pruning) linear speedup is achieved on up to 16 processors and then speedup drops off somewhat for 32 processors as it becomes more difficult to maintain an ideal load balance. For minimum support levels of 100 and 200 we observe super linear speed up. In this case, as p grows, the parallel runtime benefits from doing more and more computation in memory rather than using the disks.

[1] Apparently a number of local banks perform automated site scraping on the obituary pages looking for deceased clients.



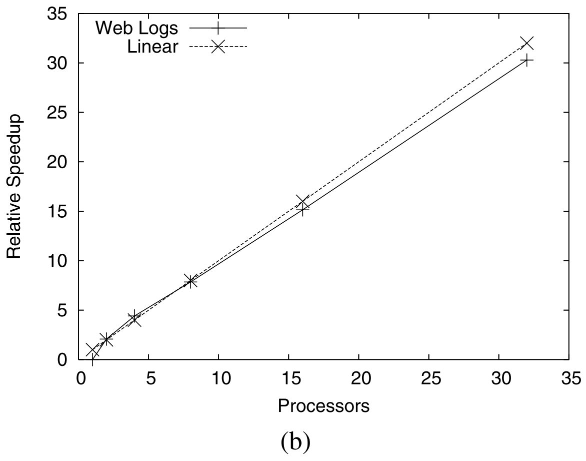


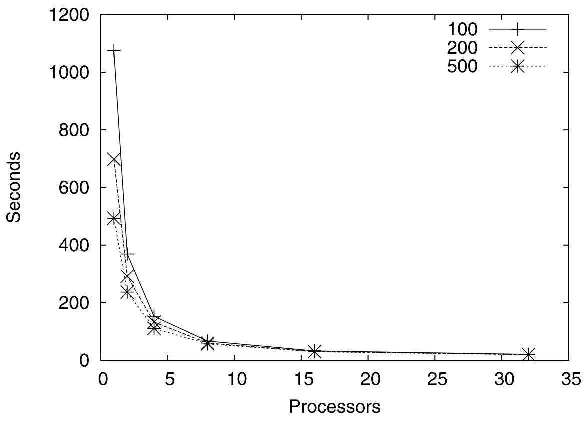
Fig. 18 Parallel external memory PnP on the News Web Log data set. (**a**) Parallel wall clock time in seconds as a function of the number of processors and (**b**) corresponding speedup. (Fixed parameters: $n = 32,631,392$, $d = 4$, $|D_i| = 833471, 366, 24, 25$, $m = 100$, $M = 100$ Mb)

One of the most important differences between the "real" data sets used in this section and the synthetic data sets used previously is that the real data sets contain complex data skew. A clear advantage of the PnP approach is that it performs well under this real world condition. In all of our experiments on real data sets PnP exhibited near linear, or even super linear, speedup.

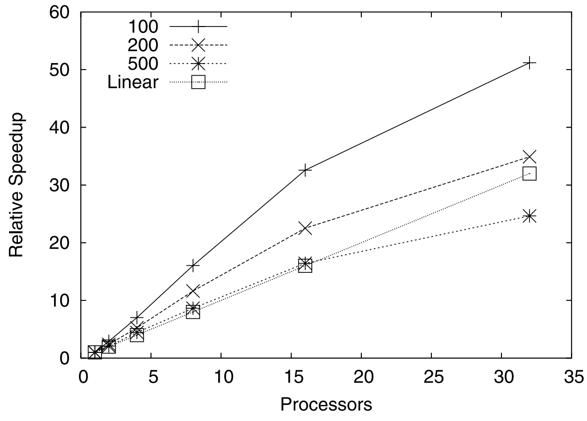
3.5 Parallel experiments on very large data sets

Finally, in this section we evaluate the performance of PnP using very large synthetic data sets. Figure 20 shows the running time for generating full data cubes from large data sets of up to 250 million rows and the running time as a function of the corresponding output data (up to one Terabyte). We observe that the running time is almost linear in terms of both input data size and output data size. Note that for data cubes of almost 1 Terabyte in size (980 Megabytes) the running time is just under 115 minutes. Overall, PnP running on a 32 processor cluster is able to build data cubes at a rate of more than half a terabyte per hour.

Fig. 19 Parallel external memory PnP on the World Hydrologic data set. (**a**) Parallel wall clock time in seconds as a function of the minimum support and (**b**) corresponding relative speedup. (Fixed parameters: $n = 17,845,529$, $d = 8$, $|D_i| = 360, 360, 180, 90, 82, 8, 7, 6$, $M = 100$ Mb)



(a)



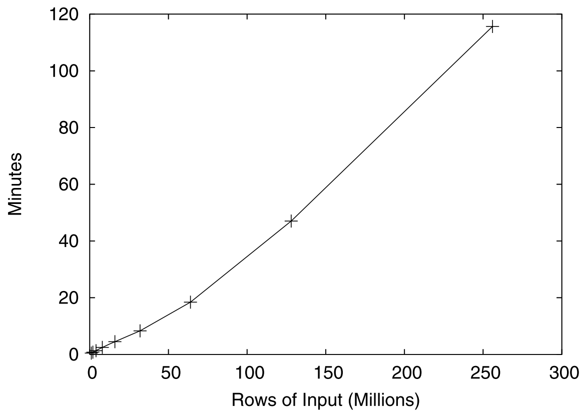
(b)

4 Conclusions

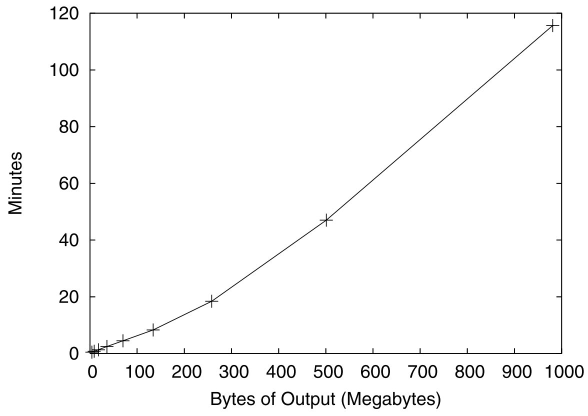
In this paper, we discussed the use of hybrid approaches for the computation of iceberg-cube queries and presented a new hybrid method for iceberg-cube query computation termed "Pipe 'n Prune" (PnP). The most important feature of our approach is that it completely interleaves top-down data aggregation through piping with bottom-up a priori data reduction.

We performed an extensive performance analysis of PnP for a variety of scenarios. For sequential iceberg-cube queries, BUC and StarCube have ranges of data density and skew where BUC outperforms StarCube or vice versa. In both cases, PnP is close to the best one which makes PnP an interesting new alternative method in applications where performance stability over a wide range of input parameters is important. For external memory iceberg-cube queries, we observed minimum loss of efficiency when PnP runs out of main memory. More precisely, the measured external memory running time for PnP is less than twice the running time for full in-memory

Fig. 20 Parallel external memory PnP on a very large data set. (**a**) Parallel wall clock time in minutes as a function of input data size and (**b**) the corresponding output data size. (Fixed parameters: $d = 8$, $|D_i| = 256$ for $1 \leq i \leq d$, $p = 32$, $M = 300$ Mb)



(a)



(b)

computation of the same iceberg-cube query. For parallel iceberg-cube queries on shared-nothing PC clusters, PnP scales well and provides near linear speedup for larger numbers of processors. In general, PnP performs very well for both, dense and sparse data sets. PnP is very efficient on real data sets and particularly well suited for those data sets that are hard to handle using sequential methods because of their large size, high dimensionality or large cardinalities.

References

1. Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J., Ramakrishnan, R., Sarawagi, S.: On the computation of multidimensional aggregates. In: Proceedings of the 22nd International VLDB Conference, pp. 506–521 (1996)
2. Beyer, K., Ramakrishnan, R.: Bottom-up computation of sparse and iceberg cubes. In: Proceedings of the 1999 ACM SIGMOD Conference, pp. 359–370 (1999)

3. Chen, Y., Dehne, F., Eavis, T., Rau-Chaplin, A.: Building large ROLAP data cubes in parallel. In: Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS '04) (2004)
4. Chen, Y., Dehne, F., Eavis, T., Rau-Chaplin, A.: Parallel ROLAP data cube construction on shared-nothing multiprocessors. Distributed Parallel Databases **15**, 219–236 (2004)
5. Codd, E.F.: Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate. Technical report, E.F. Codd and Associates (1993)
6. Dehne, F., Eavis, T., Hambrusch, S., Rau-Chaplin, A.: Parallelizing the datacube. In: International Conference on Database Theory (2001)
7. Dehne, F., Eavis, T., Rau-Chaplin, A.: A cluster architecture for parallel data warehousing. In: International Conference on Cluster Computing and the Grid (CCGRID 2001) (2001)
8. Dehne, F., Eavis, T., Rau-Chaplin, A.: Computing partial data cubes for parallel data warehousing applications. In: Euro PVM/MPI 2001 (2001)
9. Dehne, F., Eavis, T., Rau-Chaplin, A.: Parallelizing the datacube. Distributed Parallel Databases **11**(2), 181–201 (2002)
10. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)
11. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.: Computing iceberg queries efficiently. In: Proceedings VLDB, pp. 299–310 (1998)
12. Goil, S., Choudhary, A.: High performance OLAP and data mining on parallel computers. J. Data Min. Knowl. Discov. (4) (1997)
13. Goil, S., Choudhary, A.: High performance multidimensional analysis of large datasets. In: Proceedings of the First ACM International Workshop on Data Warehousing and OLAP, pp. 34–39 (1998)
14. Goil, S., Choudhary, A.: A parallel scalable infrastructure for OLAP and data mining. In: International Database Engineering and Application Symposium, pp. 178–186 (1999)
15. Gray, J., Bosworth, A., Layman, A., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In: Proceedings of the 12th International Conference on Data Engineering, pp. 152–159 (1996)
16. Han, J.: Software download site. http://www-sal.cs.uiuc.edu/hanj/pubs/software.htm
17. Harinarayan, V., Rajaraman, A., Ullman, J.: Implementing data cubes. In: Proceedings of the 1996 ACM SIGMOD Conference, pp. 205–216 (1996)
18. HYDRO1k Elevation Derivative Database. http://edcdaac.usgs.gov/gtopo30/hydro/index.asp. Last visited: Oct. 25th, 2006
19. Lakshmanan, L.V.S., Pei, J., Han, J.: Quotient cube: How to summarize the semantics of a data cube. In: Proceedings of the 28th VLDB Conference (2002)
20. Lakshmanan, L.V.S., Pei, J., Zhao, Y.: QC-trees: An efficient summary structure for semantic OLAP. In: Proceedings of the 2003 ACM SIGMOD Conference, pp. 64–75 (2003)
21. Lu, H., Yu, J.X., Feng, L., Li, X.: Fully dynamic partitioning: Handling data skew in parallel data cube computation. Distributed Parallel Databases **13**, 181–202 (2003)
22. Muto, S., Kitsuregawa, M.: A dynamic load balancing strategy for parallel datacube computation. In: ACM 2nd Annual Workshop on Data Warehousing and OLAP, pp. 67–72 (1999)
23. Ng, R., Wagner, A., Yin, Y.: Iceberg-cube computation with PC clusters. In: Proceedings of 2001 ACM SIGMOD Conference on Management of Data, pp. 25–36 (2001)
24. Ross, K., Srivastava, D.: Fast computation of sparse data cubes. In: Proceedings of the 23rd VLDB Conference, pp. 116–125 (1997)
25. Roussopoulos, N., Kotidis, Y., Roussopolis, M.: Cubetree: Organization of the bulk incremental updates on the data cube. In: Proceedings of the 1997 ACM SIGMOD Conference, pp. 89–99 (1997)
26. Sarawagi, S., Agrawal, R., Gupta, A.: On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California (1996)
27. Sismanis, Y., Deligiannakis, A., Roussopolos, N., Kotidis, Y.: Dwarf: Shrinking the petacube. In: Proceedings of the 2002 ACM SIGMOD Conference, pp. 464–475 (2002)
28. Wang, W., Feng, J., Lu, H., Yu, J.X.: Condensed cube: An effective approach to reducing data cube size. In: Proceedings of the International Conference on Data Engineering (2002)
29. Winter Corporation. 2005 Top Ten Program Summary: The survey of the world's largest databases. http://www.wintercorp.com/WhitePapers/WC_TopTenWP.pdf. Last visited Oct. 25th 2006
30. Xin, D., Han, J., Li, X., Wah, B.W.: Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In: Proceedings Int. Conf. on Very Large Data Bases (VLDB'03) (2003)
31. Zhao, Y., Deshpande, P., Naughton, J.: An array-based algorithm for simultaneous multi-dimensional aggregates. In: Proceedings of the 1997 ACM SIGMOD Conference, pp. 159–170 (1997)