DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF STRATHCLYDE

MULTIDIMENSIONAL AGGREGATION IN OLAP SYSTEMS

A THESIS SUBMITTED TO
THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE FOR POSTGRADUATE STUDIES
OF THE UNIVERSITY OF STRATHCLYDE
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Nikolaos Kotsis

February 2000

© Nikolaos Kotsis 2000

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by the University of Strathclyde Regulation 3.49. Due acknowledgement must be made of the use of any material contained in, or derived from, this thesis.

Declaration

I declare that this Thesis embodies my own research work and that it is composed by myself. Where appropriate I have made acknowledgement to the work of others.

Nikolaos Kotsis

To my Family

Abstract

On-line analytical processing (OLAP) provides multidimensional data analysis to support decision making. OLAP queries require extensive computation based on aggregation along many dimensions and hierarchies. The time required to process these queries has traditionally prevented the interactive analysis of large databases and in order to accelerate query-response time, pre-computed results are often stored as *materialised views* for later retrieval. This adds a prohibitive storage overhead when applied to the whole set of aggregates, known as the *data cube.* Storage space and computation time can be significantly reduced by partial computation.

The challenge in implementing the data cube has been to select the minimum number of views for materialisation, while retaining fast query response time.

This thesis makes significant contributions to this area by introducing the *Low Redundancy* (L-R) approach which provides the means for the selection, computation and storage of non-redundant aggregates.

Firstly, through the introduction of a novel technique, redundant aggregates are identified thus allowing only distinct aggregates to be computed and stored.

Secondly, further redundancy is identified and eliminated using a second novel technique which stores these distinct aggregates in a compact differential form.

Novel algorithms were introduced to implement these techniques and provide a solution which is both scalable and low in complexity.

Both techniques have been evaluated using real and synthetic datasets with experimental results, and have achieved significant savings in computation time and storage space compared to the conventional approach. Savings have been shown to increase as dimensionality increases.

Existing techniques for implementing the data cube differ from the L-R approach but they can be integrated with it to achieve faster query-response time.

Finally, the implications of this work reach beyond the area of OLAP to the fields of decision support systems, user interfaces and data mining.

Acknowledgements

I would like to thank my supervisor Douglas McGregor for his financial support and advice during my thesis, especially during his absence caused by serious illness. I am glad that he was able to make a complete return to work and I wish him continued good health.

My special thanks to Andrew McGettrick for acting as a temporary supervisor for a year and a half. I also thank him for his advice and support thereafter.

I would also like to acknowledge George Weir, Mohamed Ould-Khaoua and Akis Petropoulakis for their valuable advice and discussions during my research and especially for reading this thesis.

Richard Fryer, Andrew Forrest, yvind Stromme, Antoan Izmirlief and Robert Lambert were great company during my studies and it was my pleasure to be involved with them in different research activities.

I am grateful to my family for the support they have given me not only during my PhD degree but throughout my education and life. I thank my brother Thomas for his encouragement and interest during my studies and for helping me to find the right path. I also thank my brother Kostis for his enthusiasm about my work. They both gave me strength and motivation. Last but not least, I thank my parents for their endless help and trust in my decisions throughout my life and studies.

My greatest thanks goes to my wife Geraldine for her constant patience and understanding during the long hours of my studies. With admirable strength and ability, she undertook the strain of managing everything else to allow me to stay focused on my work. This thesis would not have started or finished without her.

Contents

Declaration ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 The Traditional Relational Database Model 3

1.1.1 Relational Algebra and SQL 4

1.1.2 Redundancy in Existing Relational Theory 5

1.2 The Multidimensional Conceptual Model 5

1.2.1 The ROLAP Model 7

1.3 The Aggregate Functions 9

1.4 Main Issues in the Implementation of the Data Cube 10

1.5 Approaches to the Implementation of the Data Cube 12

1.5.1 Trade-off between Time and Space 13
1.6 Contributions 14
1.7 Thesis Outline 16
2 Background **18**
2.1 The Cube-by Operator 18
2.2 The Set of Aggregations: A Hypercube Lattice 19
2.2.1 The Dimension Hierarchy Lattice 21
2.3 The Role of Materialised Views in Data Warehousing 22
2.3.1 Related Work 24
2.3.2 Methods for Aggregation 24
2.3.3 Computing the Data Cube 25
2.3.4 Selection of Views for Materialisation 27
2.3.5 Further Related Work 29
3 The Low Redundancy Concept **31**
3.1 Totally-Redundant Views 32
3.1.1 Extending the Relational Theory: Totally-Redundant Views 33
3.1.2 Example of Totally-Redundant Views 35
3.2 Partially-Redundant Views 35
3.2.1 Extending the Relational Theory: Partially-Redundant Views 36
3.2.2 Example of Partially-Redundant Views 37
3.3 Difference Algebraic Equations 37
3.4 Implementation Considerations 39
3.4.1 View Types 40
3.5 Implementing the Totally-Redundant views 42
3.5.1 The Key Algorithm 42

3.5.2 The Complexity and Performance of the Key Algorithm 43
3.5.3 The Recursive Key Algorithm 46
3.6 Implementing the Partially-Redundant Views 47
3.6.1 The Aggregation Algorithms 47
3.6.2 The B-Aggregator 48
3.6.3 The V-Aggregator 49
3.6.4 Example of the V-Aggregator 49
3.7 Computing the L-R Data Cube 50

4 Experimental Confirmation 54

4.1 The Experimental Configuration 55
4.1.1 The Datasets 55
4.2 Computing the Data Cube - Performance Timings 57
4.2.1 The Performance of the Key algorithm 57
4.2.2 Full Computation of the Data Cube 59
4.3 Storage of the Materialized Views - space savings 62
4.4 Totally-Redundant Views of Derivative Relations 64
4.5 Query Response Time - Performance Timings 65

5 Conclusion 68

5.1 Implications of the L-R approach 70
5.1.1 Indexing in OLAP 70
5.1.2 The User Interface 71
5.1.3 The Main Store 72
5.2 Future Work 73

Bibliography 75

Appendices **82**

A Source code: L-R aggregation **83**

B Source code: L-R data cube **90**

C Analytical Results **114**

C.1 Computing the Data Cube 114

C.2 Storage of Materialised Views 122

C.3 Totally-Redundant Views in Derivative Relations 130

D The Semi-Join **138**

List of Figures

1.1 A Three Dimensional tuple (Product, Time, Location, Sales) 7
1.2 The Star schema 8
1.3 The Location hierarchy 8
1.4 Data cube size (in tuples) vs dimensions 11
1.5 Implementing the data cube 12
1.6 The curve of benefit 14

2.1 The Hypercube lattice 20
2.2 The dimensions hierarchy 21

3.1 The g-equivalent tuple 33
3.2 Producing the Difference representation 38
3.3 The types of views in L-R 41
3.4 The cube lattice 43
3.5 The performance of the Key algorithm 44
3.6 Performing an aggregation using the B-Aggregator 48
3.7 Extracting the differences during aggregation using the V-Aggregator . . . 51
3.8 The L-R data cube 52

4.1 The conventional data cube time compared to the Key-algorithm time in the TPC-D 600K dataset . . . 58
4.2 The conventional data cube time compared to the Key-algorithm time in the Hotel dataset . . . 58
4.3 Time required to compute the data cube: conventionally over L-R, average of the six datasets . . . 61
4.4 Performance in time of the L-R approach compared to the conventional approach in different datasets . . . 61
4.5 Space savings growth ratio in all datasets . . . 63
4.6 Performance in space savings of the L-R approach compared to the conventional approach in different datasets . . . 63
4.7 Group-by's response time . . . 67

5.1 Elimination of Totally-Redundant views . . . 72

C.1 Time performance of the L-R approach in TPC-D Lineitem Table (600K) . 116
C.2 Time performance of the L-R approach in TPC-D Lineitem Table (60K) . . 117
C.3 Time performance of the L-R approach in TPC-D Lineitem Table (6K) . . 118
C.4 Time performance of the L-R approach in Hotel dataset . . . 119
C.5 Time performance of the L-R approach in the Weather dataset . . . 120
C.6 Time performance of the L-R approach in the Adult dataset . . . 121
C.7 Space savings of the L-R approach in TPC-D Lineitem Table (600K) . . . 124
C.8 Space savings of the L-R approach in TPC-D Lineitem Table (60K) . . . 125
C.9 Space savings of the L-R approach in TPC-D Lineitem Table (6K) . . . 126
C.10 Space savings of the L-R approach in the hotel dataset . . . 127
C.11 Space savings of the L-R approach in the weather dataset . . . 128
C.12 Space savings of the L-R approach in the Adult dataset . . . 129

C.13 The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (600K) 132
C.14 The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (60K) 133
C.15 The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (6K) 134
C.16 The effect of Totally-Redundant views on space in the Hotel dataset 135
C.17 The effect of Totally-Redundant views on space in the Weather dataset . . 136
C.18 The effect of Totally-Redundant views on space in the Adult dataset 137

List of Tables

1.1 Comparison between OLTP and OLAP applications 2

1.2 The *Customer* relation 4

1.3 The *Sales_Transaction* Relation 6

1.4 The *Aggregate* Relation 10

2.1 The *Sales_Transaction* Relation 19

3.1 Notation of main components 32

3.2 The Input Relation R 36

3.3 The g-equivalent aggregate relation R' of relation R (Table 3.2) 36

C.1 Time performance of the L-R approach in TPC-D Lineitem Table (600K) . 116

C.2 Time performance of the L-R approach in TPC-D Lineitem Table (60K) . . 117

C.3 Time performance of the L-R approach in TPC-D Lineitem Table (6K) . . 118

C.4 Time performance of the L-R approach in Hotel dataset 119

C.5 Time performance of the L-R approach in the Weather dataset 120

C.6 Time performance of the L-R approach in the Adult dataset 121

C.7 Space savings of the L-R approach in TPC-D Lineitem Table (600K) 124

C.8 Space savings of the L-R approach in TPC-D Lineitem Table (60K) 125

C.9 Space savings of the L-R approach in TPC-D Lineitem Table (6K) 126

C.10 Space savings of the L-R approach in the hotel dataset 127
C.11 Space savings of the L-R approach in the weather dataset 128
C.12 Space savings of the L-R approach in the Adult dataset 129
C.13 The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (600K) 132
C.14 The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (60K) 133
C.15 The effect of Totally-Redundant views on space in TPC-D Lineitem Table (6K) 134
C.16 The effect of Totally-Redundant views on space in the Hotel dataset 135
C.17 The effect of Totally-Redundant views on space in the Weather dataset . . 136
C.18 The effect of Totally-Redundant views on space in the Adult dataset 137

Chapter 1

Introduction

Technological developments which assist in the abstraction of useful insights from large volumes of data are becoming increasingly important today as industrial, commercial and scientific databases proliferate and grow in volume. This thesis introduces novel theory and system's design that greatly contribute to one such area - the provision of multi-dimensional aggregates in *On-Line-Analytical-Processing (OLAP)*. OLAP tools provide multidimensional data analysis by computing summaries and breakdowns along many dimensions [FSS95]. They are designed for decision support where historical, summarised and consolidated data is more important than detailed, individual records [CD97].

The functional and performance requirements of OLAP systems differ from those of *On-Line-Transaction-Processing (OLTP)* which were traditionally supported by operational databases [Codd93]. OLTP applications typically automate the day-to-day operations such of an organization as clerical data processing tasks, order entry and banking transactions. These applications are structured, repetitive and consist of short, atomic, isolated transactions. Thus, while an OLTP application needs to record details of an individual transaction, an OLAP application provides analysis of consolidated information about large numbers of transactions. OLAP queries are complex, read-only queries, in

contrast to those posed in OLTP systems which usually deal with less complex read/write queries. Table 1.1 summarises the differences between OLTP and OLAP database applications [Sch97].

	OLTP	**OLAP**
Data	atomic	summarised
Usage of system	run business	analyse business
User interaction	pre-determined	ad-hoc
Work characteristics	read/write	read mostly
Typical user	clerical	professional
Unit of work	transaction	query
Records accessed	tens	millions
Number of users	thousands	hundred
Focus	data in	info out

Table 1.1: Comparison between OLTP and OLAP applications

Decision support systems (DSS) provide large-scale data analysis with facts from previously stored, historical data. *Data warehouses*, through their integrated collection of data, provide the infrastructure for *DSS* applications [Wid95], [BZ98]. OLAP tools, as a part of *DSS*, are designed to provide multidimensional breakdowns involving large numbers of aggregate queries on detailed data [Gm99]. Aggregation operates by grouping records belonging to a specified set of domains. In line with [John98], an aggregate is a function - *count, sum, average, maximum, minimum* - which operates on some specific column of a relation and is applied separately inside each set of grouping attributes.

OLAP queries may have to process millions, if not billions, of records in a data warehouse, which increases the processing cost. The key problem is that the number of possible aggregates in a database can be large and the time required to process any of these aggregates 'on the fly' as a part of an interactive dialogue is prohibitive.

One approach to accelerate the querying response time is to pre-compute and store the results in advance for later retrieval. However, for large relations with a large num-

ber of dimensions full pre-computation and storage increases storage overheads and the processing time required to compute all the aggregations.

The proposed *Low-Redundancy (L-R)* approach introduced in this thesis is a novel way of achieving fast computation and compact storage of the aggregates, through the selection of non-redundant aggregates. This is achieved by extending relational theory and applying it to the OLAP environment. The L-R approach differs from existing techniques for selecting and storing the multidimensional aggregates [HRU96], [SDN98], [BPT97], [BR99]. This novel approach is however compatible, and could be combined, with these techniques.

1.1 The Traditional Relational Database Model

The main structure of the relational model is the *relation*. Following the mathematical description of Codd [Codd77], given the sets $S_1, S_2, ..., S_n$, R is a relation on these n sets if it is a subset of the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$. In line with [Ram98], a relation consists of a *relation schema* and a *relation instance*. The relation schema describes the relation name, the name of its columns (fields or attributes) and the domain of each column. The following example shows a relation schema for the relation *Customer* shown in Table 1.2:

*Customer(Name:***String***, Address:***String***, Age:***Integer***, TelephoneNo:***Integer***)*

The above schema indicates that the attributes Name, Address, Age and TelephoneNo have domains named String, String, Integer and Integer respectively.

A *relation instance* is a set of tuples, also called records, in which each tuple has the same number of fields as the relation schema.

Customer			
Name	Address	Age	TelephoneNo
C1	Ad1	34	4506632
C2	Ad2	23	3050312
C3	Ad3	45	2242342
C4	Ad4	56	8202356
C5	Ad2	25	9657302
C6	Ad4	23	5034366
C7	Ad5	42	4524504
C8	A5	33	9855652
C9	Ad6	34	3380867

Table 1.2: The *Customer* relation

When a domain (or combination of domains) of a given relation has values which uniquely identify each element (n-tuple) of that relation it is called a *candidate key.* A candidate key is non-redundant and is either a single domain or a combination such that none of the participating domains are superfluous in uniquely identifying each tuple. A relation may have more than one candidate key and when this occurs, one of them is arbitrarily selected and called the *primary key.*

1.1.1 Relational Algebra and SQL

A database can be accessed using two formal languages - relational algebra and relational calculus [Ram98]. Relational algebra allows the user to compose operators to form a complex query through its *relational algebra expressions.* Operators such as *selection, projection, union, cross-product and difference* can be expressed in relational algebra as well as *join* and *division.* Relational calculus provides a declarative, non-procedural language in which users can express the answer of interest. However, their mathematical notation, relational algebra and relational calculus makes them unsuitable for non-technical users. Another language, the structured query language (SQL), is more appropriate for a wider

audience. A basic form of an SQL syntax with its intepretation is as follows:

SELECT *(projected attributes)*

FROM *(relation name)*

WHERE *(selected tuples)*

GROUP BY *(aggregate attributes)*

1.1.2 Redundancy in Existing Relational Theory

Codd [Codd70] referred to the redundancy in the named set of relations and the stored set of relations and identified two categories as follows:

- *Strong Redundancy.* A set of relations is strongly redundant if it contains at least one relation that possesses a projection which is derivable from other projections of relations in the set.
- *Weak Redundancy.* A collection of relations is weakly redundant if it contains a relation that has a projection which is not derivable from other members but is at all times a projection of some join of other projections of relations in the collection.

These methods for redundancy may be useful in traditional relational system's design but not in OLAP where the main problem is the expansion of data in the formation of the data cube. This will be discussed later in this chapter.

1.2 The Multidimensional Conceptual Model

The motivation behind the multidimensional approach is the need to describe complex OLAP queries in an intuitive way. The multidimensional conceptual model has been adopted fairly widely as an alternative to the relational conceptual model for OLAP ap-

plications. Consider the *Sales_Transaction* relation in Table 1.3 with three attributes *Product, Location* and *Time.*

Sales_Transaction			
ProductID	LocationID	TimeID	Sales
P1	L1	20/01/99	50
P1	L1	20/01/99	34
P1	L2	03/03/96	22
P2	L3	16/10/98	8
P2	L3	16/10/98	96
P2	L1	20/01/99	56
P2	L1	09/04/95	45
P3	L2	26/02/97	98
P3	L2	26/02/97	33

Table 1.3: The *Sales_Transaction* Relation

Typically, the user is interested in aggregating an attribute of interest called the *measure* (e.g., the attribute *Sales* in Table 1.3). The relationship between the measure and other attributes in a relation can be realised in the multidimensional domain when attributes on which the measure depends are considered to be *dimensions.* This transformation from the relational to multidimensional model may be represented as a Hypercube [HRU96]. Figure 1.1 shows the tuple *(P2, L1, 09/04/95, 45)* as a point in the three dimensional space, with coordinates being the values of the dimension attributes. This is a simple example of a transformation from the relational to multidimensional model.

When the underlying structure of data is organised as relations, the approach is called *Relational OLAP* or *ROLAP* [CD97], [MUW99]. When the physical structure for OLAP databases is a multidimensional cube the approach is called *Multidimensional OLAP* or *MOLAP.* In the latter approach, the physical data storage in multidimensional arrays corresponds to the conceptual multidimensional model and OLAP queries can then be answered directly.

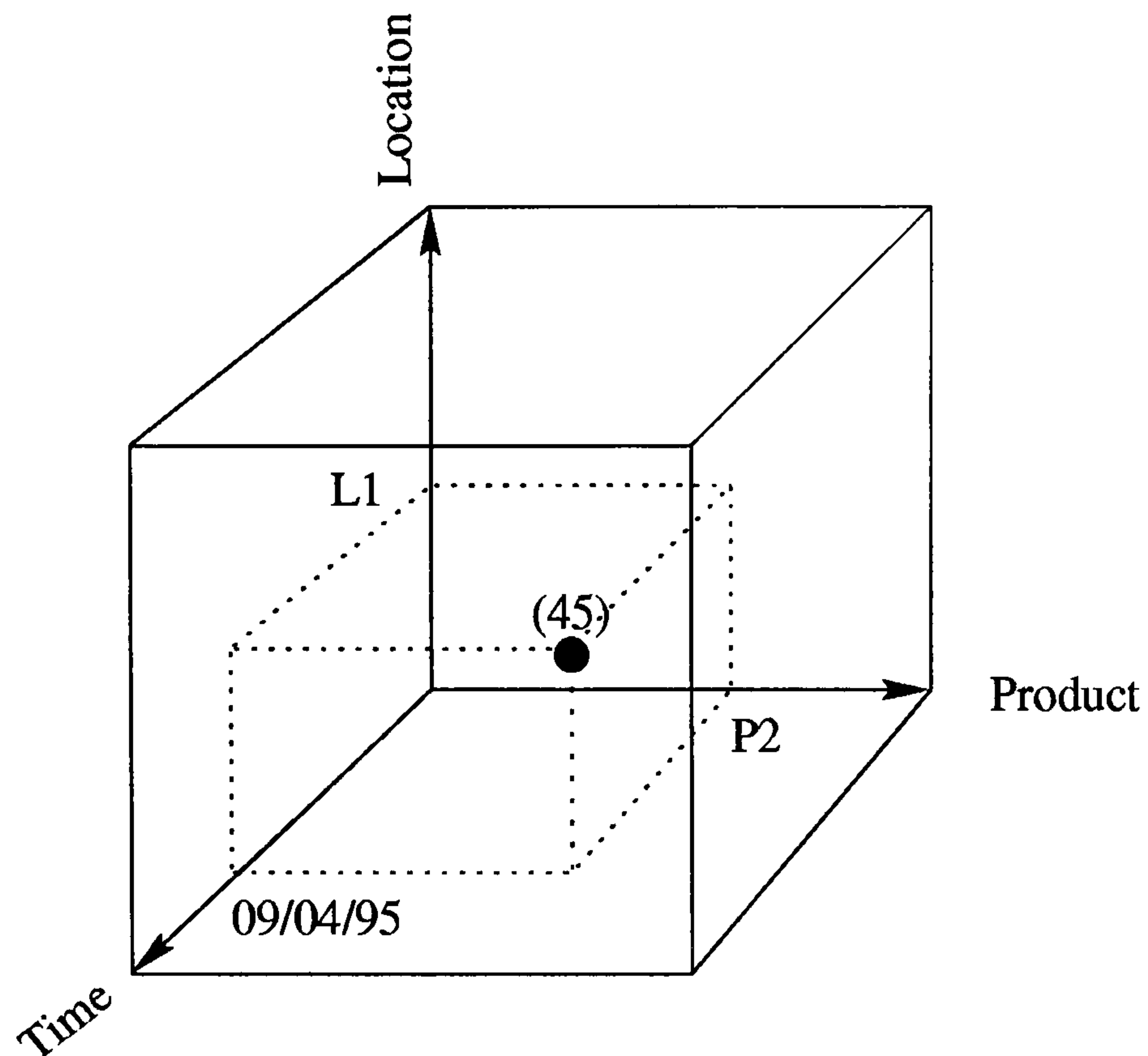

Figure 1.1: A Three Dimensional tuple (Product, Time, Location, Sales)

1.2.1 The ROLAP Model

The typical data organization in a ROLAP model is to store the detailed data in a table known as the *fact table* (as shown in Table 1.3) and any information related to the detailed data in separate tables known as the *dimension tables*. This structure is called the *star schema* [Kim96]. An example of a star schema is shown in shown in Figure 1.2 with the following schema for the fact table:

Sales_Transaction(ProductID, LocationID, TimeID, Sales)

and the following dimension tables:

Product(ProductID, Type, Category)

Location(LocationID, City, Country, Continent)

Time(TimeID, Month, Year)

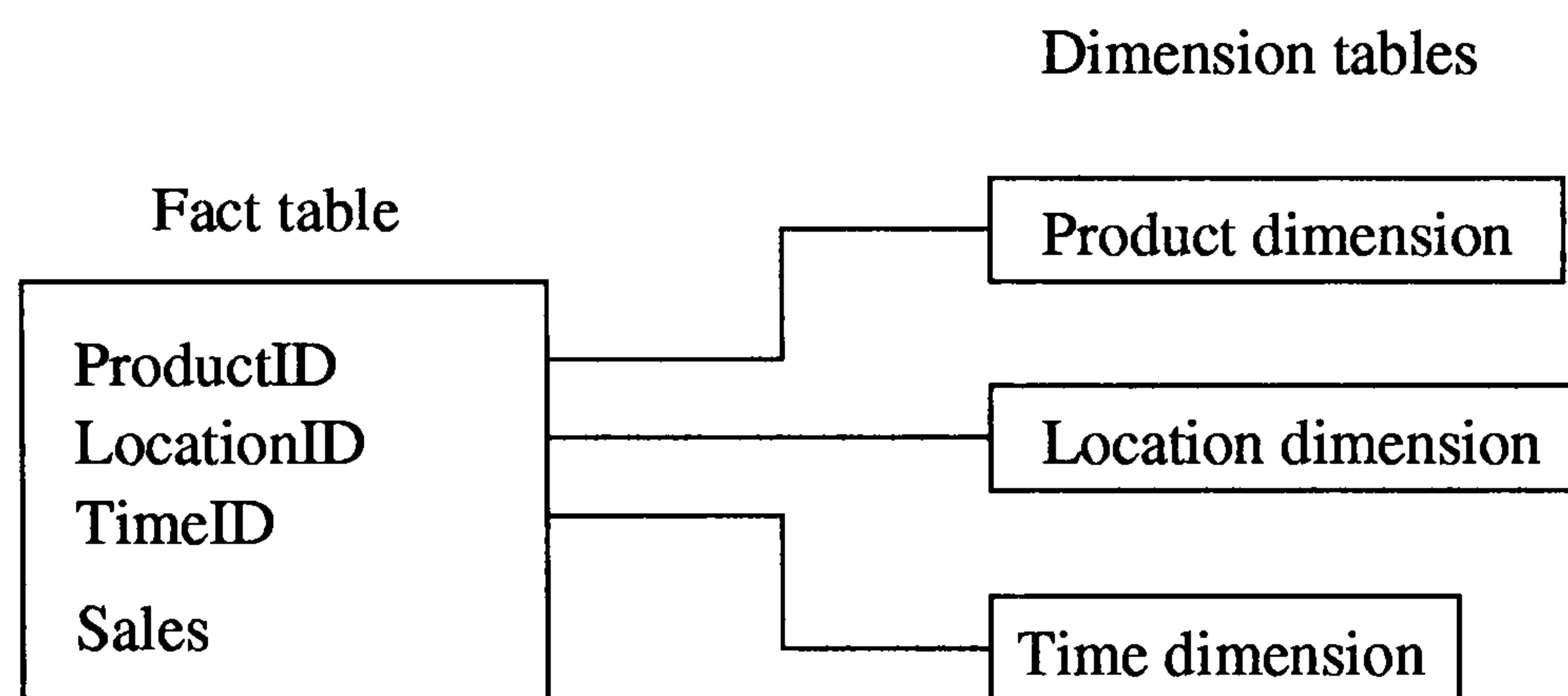

Figure 1.2: The Star schema

The relation *Sales_Transaction* contains a tuple for every product sold in a transaction. Thus, the fact table shown in Table 1.3 contains three dimensions and a measure of interest, namely *Sales*. Each dimension may have a set of attributes denoting the hierarchy in this dimension. A simple example of a hierarchy is the LocationID, in the *Location* dimension schema, in which *City* belongs to *Country* and *Country* belongs to *Continent*. This hierarchy is illustrated in Figure 1.3.

LocationID → **City** → **Country** → **Continent**

Figure 1.3: The Location hierarchy

Navigation through different levels of summary information is achieved by operations such 'drill-down' and 'roll-up'. The drill-down operation is the process of examining the data from the abstract to a more detailed level of the hierachy. As the drilling progresses more detailed information is revealed e.g., from Continent to Country to City to particular location (LocationID). The opposite operation is called roll-up in which the examination of data moves from a more detailed to abstract level.

OLAP queries often require multiple joins between the fact table and the dimension tables. For example, the query *"Give me the total sales of the product P1 by Year and City"*, would require the aggregation in ProductID and the join between the fact table and the dimension tables *Location* and *Time* respectively. The dimension attributes in the fact table are foreign keys of the corresponding dimension tables. When dimension tables are further normalised to reduce redundancy, the resulting data organisation is referred to as a *snowflake schema* [CD97].

1.3 The Aggregate Functions

Aggregations are classified into *scalar aggregates* and *aggregate functions* [Gra93]. Scalar aggregates calculate a single scalar value from an unary input relation, e.g., the maximum value of an attribute of a relation. Users often seek information of larger granularity, e.g., *Sales* per *Product* and *Time*, irrespective of the *Location*. This aggregation of *Sales* over the *Product* and *Time* dimensions can be expressed using the following SQL statement:

SELECT *Product.ProductID*, *Time.TimeID* as *(SUM)Sales*
FROM *Sales_Transaction*
GROUP-BY *Product.ProductID*, *Time.TimeID*

The aggregation functions are relational operators; they consume and produce relations [Gra93]. An aggregation function takes a binary input relation (e.g., total of Sales in each month). The key element of the obtained new relation is the 'BY-list' or grouping attributes. Applying the above statement to the *Sales_Transaction* relation in Table 1.3 produces the derivative relation *ProductID by TimeID* shown in Table 1.4.

Product by Time		
ProductID	TimeID	Sales
P1	20/01/99	84
P1	03/03/96	22
P2	16/10/98	104
P2	20/01/99	56
P2	09/04/95	45
P3	26/02/97	131

Table 1.4: The *Aggregate* Relation

Gray et al. [GBLP96] classify Aggregate functions into the following categories :

- Distributive - Count(), Min(), Max(), Sum()
- Algebraic - Average(), Standard Deviation(), MaxN(), Min(), centre of mass()
- Holistic - Median().

Work by [Klug82] and [OOM87] presents aggregation expressions by extending the relational algebra and relational calculus.

1.4 Main Issues in the Implementation of the Data Cube

The main issue in the computation of multidimensional aggregates, known as the data cube, is the presence of large tables with large numbers of dimensions which presents a

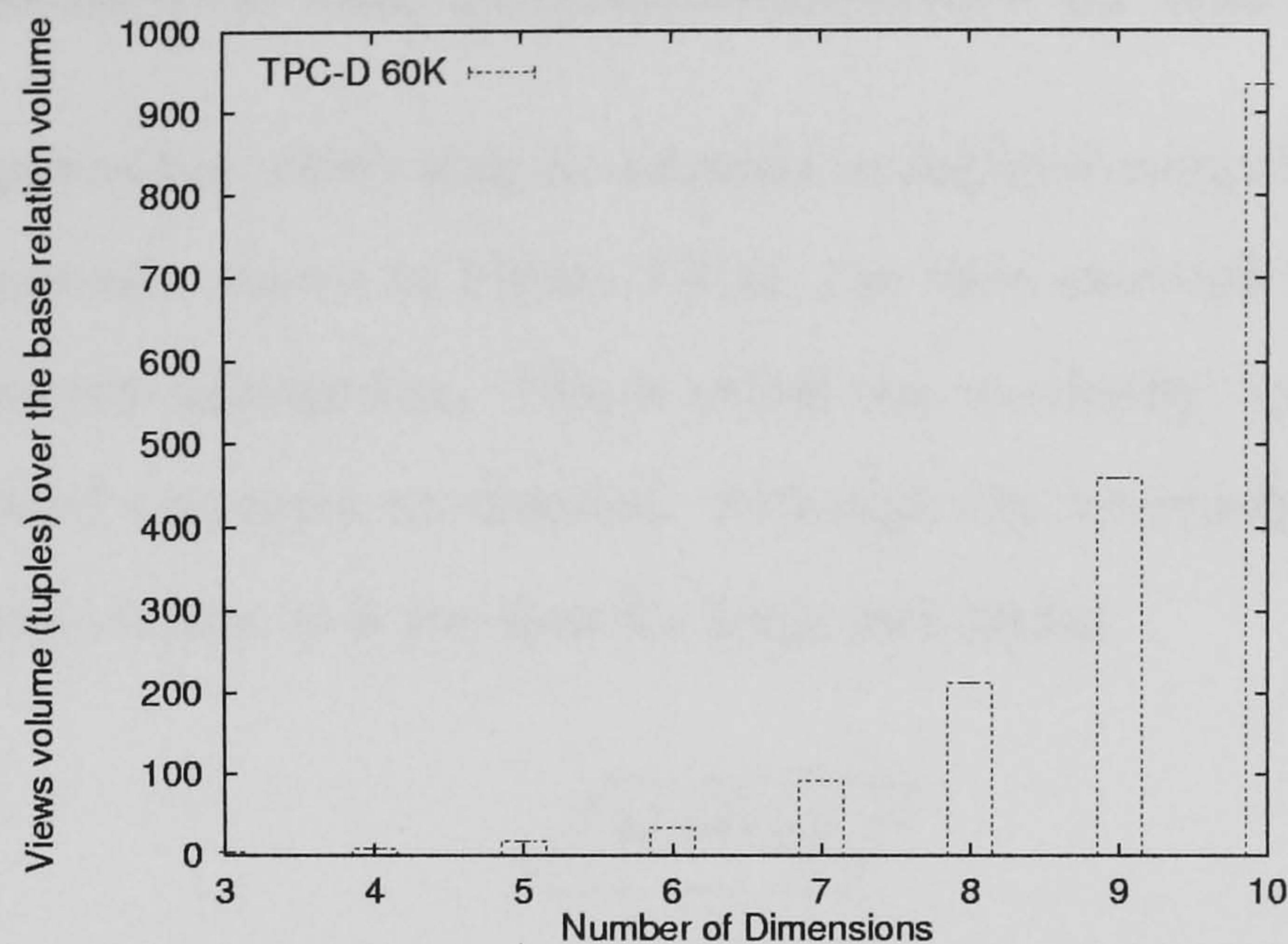

Figure 1.4: Data cube size (in tuples) vs dimensions

performance problem for the database designer. A given measure in n dimensions gives rise to 2^n possible combinations or aggregates (thus a relation with 16 attributes would require 65,536 aggregates, excluding hierarchies).

Another issue is that the fact tables in OLAP databases are usually sparse. Sparseness occurs when a table has a small cardinality (number of tuples) compared to the cross product of the cardinalities of its dimension domains. Sparseness causes the volume of materialized views to be orders of magnitude larger than the input relation. The effect of sparseness on multidimensional aggregates has been considered by [RS97], [Kim96] and [Pen99]. Figure 1.4 shows the growth (in tuples) experienced during this research work when materialising the full data cube of a ten-dimension dataset using the TPC-D benchmark dataset. The experiments demonstrate that, for the particular example, the data volume required for the datacube (using the TPC-D, 60K) was approximately two orders of magnitude greater than the base relation. The efficient implementation of the data cube is the main focus of this thesis.

1.5 Approaches to the Implementation of the Data Cube

There are three approaches which may be adopted in implementing the data cube.

In the first approach, shown in Figure 1.5(a), the data retrieval mechanism directly computes the necessary aggregation. This is called the 'on-the-fly' approach which computes every requested aggregate on demand. Although the 'on-the-fly' approach is very economical in storage terms, it is too slow for large fact tables.

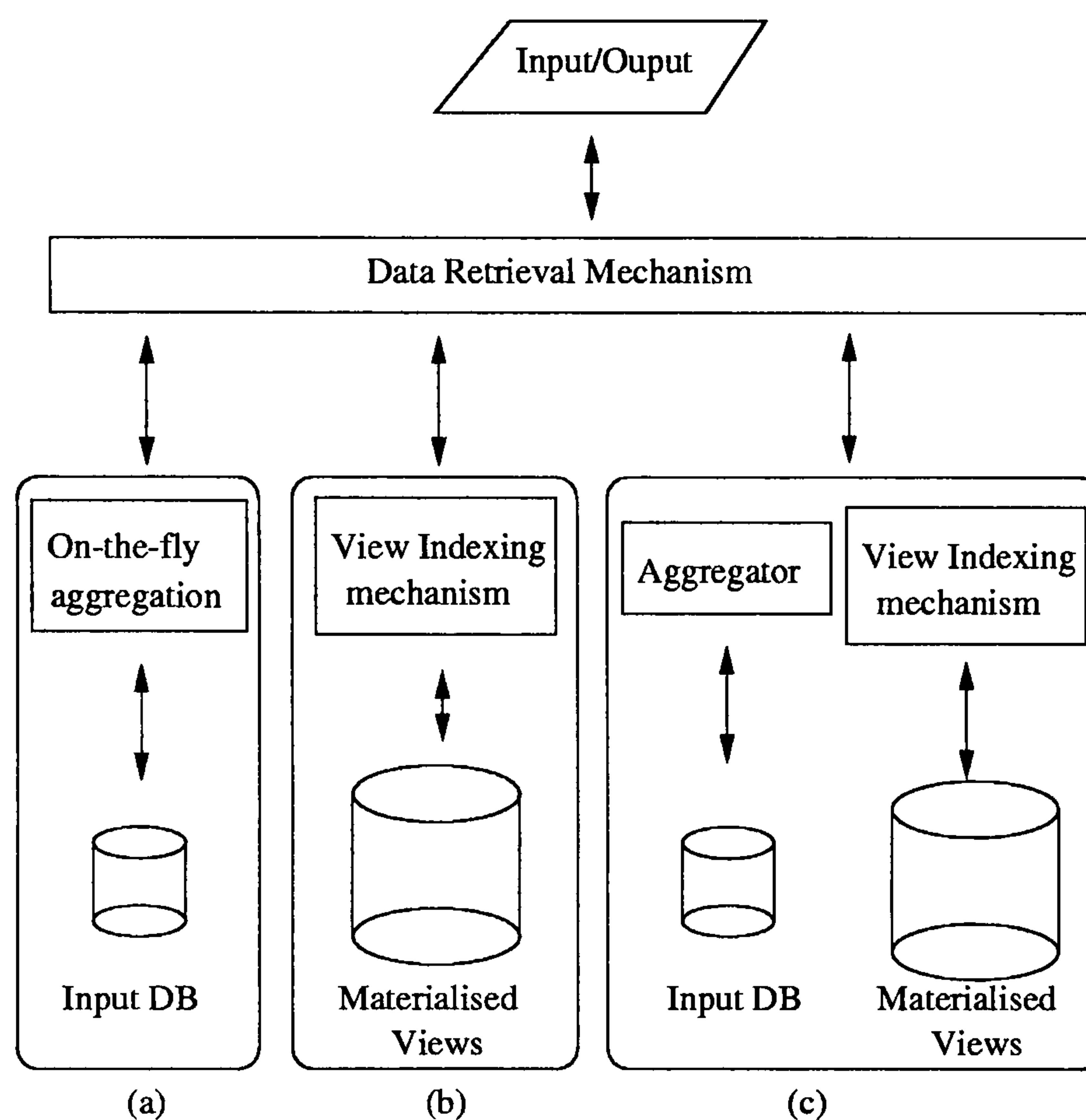

Figure 1.5: Implementing the data cube

The second approach, depicted in Figure 1.5(b), pre-computes all possible aggregates in advance and stores them as summary tables for later retrieval. These summary tables are referred to as *materialised views*. This approach has been adopted to overcome the poor response of 'on the fly' implementation. For high level aggregations, the materialised

views approach is not expensive in terms of additional storage resources and provides rapid response. The low-level abstractions, however, are large and numerous, resulting in a many-fold expansion of the original relation. Though significant research has been carried out to optimize the materialized view approach, the method encounters several difficulties mainly due to the large number of views [HRU96]. It also implies a long pre-computation time and precludes even modest updating. The experiments conducted in this thesis (refer to Chapter 4) show that full materialisation typically requires at least two orders of magnitude more space than the input base table.

Finally, the third approach attempts to select only a subset of the views for materialisation, as illustrated in Figure 1.5(c). Selecting a subset of aggregates reduces the computation time and also minimizes the space requirements. Systems which adopt this method attempt to reduce the total query response and the cost of computing the selected views, given limited amounts of resources (time and space). Existing techniques (algorithms) for the selection of materialised views have been proposed by [HRU96], [Gupt97], [BPT97] and [SDN98]. Chapter 2 presents a more detailed description of the main techniques for selection of the multidimensional aggregates.

1.5.1 Trade-off between Time and Space

The long processing time required to compute an OLAP query, forces database workers to trade space for time. Consider the schematic curve illustrated in Figure 1.6, first presented by [SDN98] which shows the problem facing the database designer. The x-axis represents the storage cost and the y-axis the time required for a database to answer an OLAP query. The horizontal dashed-line of the curve denotes the trade-off of space against time (query response) in the materialised view approach and the vertical dashed-line denotes the trade-off of time against space. The optimum solution is represented by the solid line in which the balance between time and space cost provides an economical and fast

OLAP system. The closer the curve moves to the origin the greater the balance between space and time. Achieving this optimum with existing techniques is a computationally intractable problem [SDN98] as will be shown in Chapter 2.

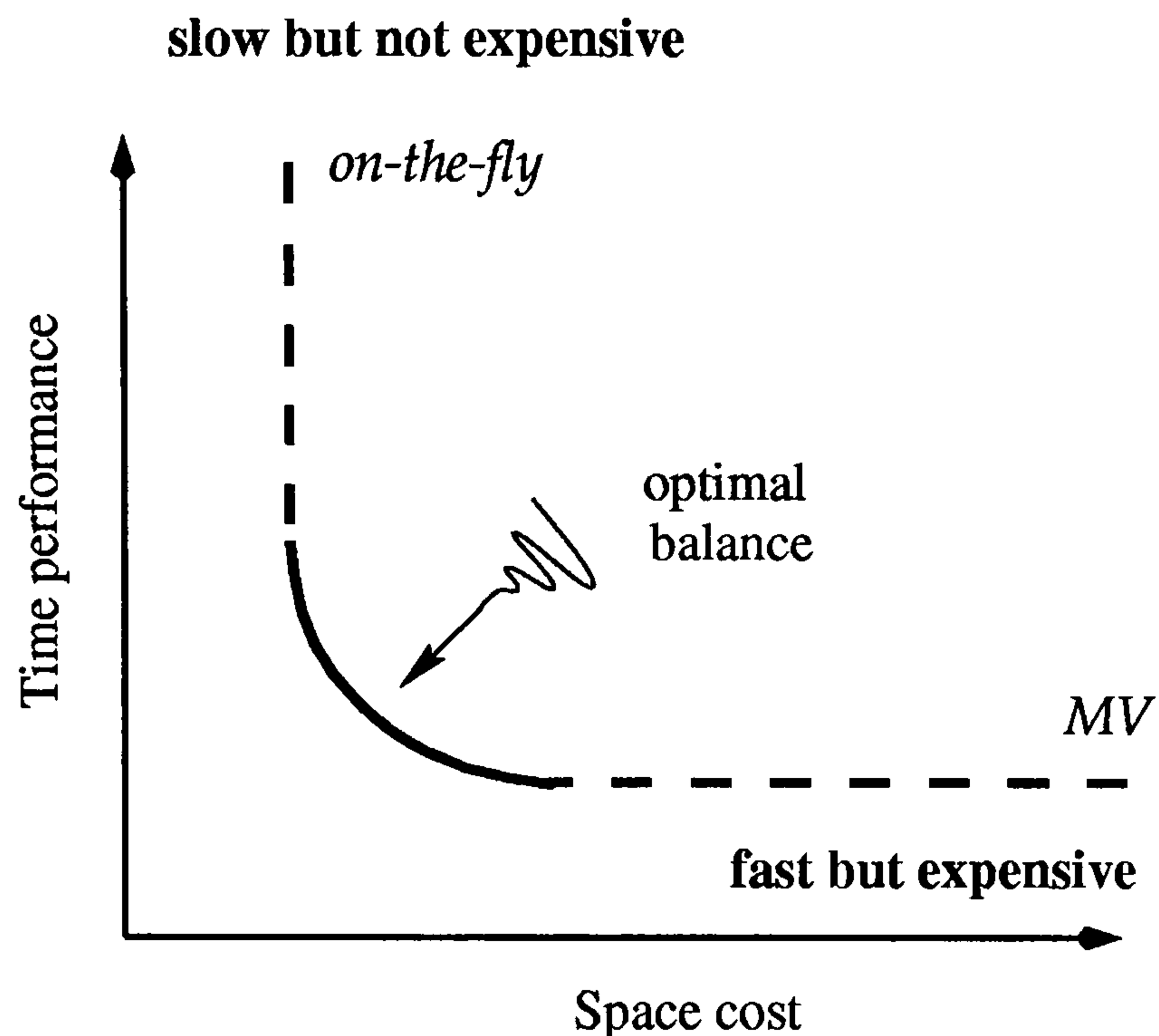

Figure 1.6: The curve of benefit

1.6 Contributions

The fundamental problem encountered by all OLAP systems when they adopt the materialized view approach is that the volume of materialized views expands exponentially with the number of dimensions. Existing techniques, described in Chapter 2, attempt to overcome the problem by selecting an appropriate subset of views for materialisation, on the basis that others can be computed more easily from the stored subset. However, no method to date has considered redundancy in the data representation and how this concept can lead to a new optimised approach.

The main contribution of this thesis is the introduction and proposition of the L-R

approach as a new efficient way of selecting and storing multidimensional aggregates in an OLAP system.

The L-R first identifies the redundant views and then computes and stores only the distinct or non-redundant ones. The whole set of aggregates can however be retrieved later in the querying process without any significant compromise in time. More specifically, the novel L-R methodology claims to:

- Select only a subset of the distinct aggregates for computation and storage on the basis that only this subset needs to be processed. The distinct aggregates can be used later to produce the full set of aggregates (the full data cube) without any additional cost. The selection algorithm requires approximately 10% of the time conventionally required to compute the data cube. As the selected distinct subset is smaller than the whole set of aggregates, the overall computation is considerably faster than the conventional method described by [GBLP96].

- Efficiently store the computed aggregates. This can be achieved by introducing a differential representation explicitly storing only those tuples which are distinct from those of the input relation. This technique can achieve remarkable savings in storage space.

- Retrieve any aggregate relation (or group-by) almost instanteneously as if it were a conventional materialised view. This performance can be achieved with a small additional cost in storage space for each aggregate relation.

Futher contributions of this thesis are:

- Proposing novel extensions to relational theory and applying it in an OLAP context. Standard relational theory does not take account of the problem of data expansion in implementing the data cube. The work presented in this thesis provides extensions to relational theory pertinent to the data cube.
- Introducing new algorithms for the selection, computation and storage of multidimensional aggregates. The proposed algorithms are scalable with low complexity.
- Providing an extensive set of experimental results confirming the theory by empirical measurements with the goal of demonstrating fairly the practicability of the new approach.

1.7 Thesis Outline

In Chapter 2, the state of the art in the field of OLAP is described. Existing methods for the selection, computation and storage of multidimensional aggregates are discussed.

Chapter 3 presents the Low-Redundancy (L-R) approach. The theoretical and practical issues of the approach are discussed. A set of algorithms is also presented as a basis for implementation techniques.

Chapter 4 details the experimental work which has been carried-out. The experiments use two real datasets, the Transaction Processing Council (TPC) benchmark dataset in three different scale factors and one synthetic dataset. These demonstrate that the theoretical advantages of the L-R approach can be achieved in practice and also that it is scalable.

Chapter 5 presents the conclusion and proposed future work. The implications of the L-R approach in the area of decision support systems, such as the user interface for OLAP,

data mining and indexing of the materialised views, are discussed. Appendix A and B present the source code which was used to evaluate the L-R approach. Appendix A shows the aggregation routine. Appendix B presents the Group-By and Cube-By objects with their methods. Appendix C presents the analytical experimental results carried-out in this thesis. Finally Appendix D discusses the conventional implementation of the Semi-join operator.

Chapter 2

Background

2.1 The Cube-by Operator

With the introduction of the *cube-by* operator by *Gray et al.*[GBLP96], all possible aggregates can be expressed in one SQL statement. The cube-by operator is presented as the n-generalisation of simple aggregate functions and the system which executes the cube-by operator has to provide all the possible aggregates. In the relation *Sales_Transaction* in Table 2.1, the following aggregations are possible: (*ProductID_LocationID*), (*ProductID_TimeID*), (*LocationID_TimeID*), (*ProductID_LocationID_TimeID*), (*ProductID*), (*LocationID*), (*TimeID*), (*All*). A given measure in n dimensions gives rise to 2^n possible combinations. Thus the cube-by operator computes every group-by corresponding to all possible combinations from a list of dimensions. In the above example, all the 2^n possible combinations can be expressed in SQL by one cube-by statement as follows:

SELECT $S.ProductID, S.LocationID, S.TimeID$ AS $(SUM) Sales$

FROM S (Sales_Transaction)

CUBE-BY $S.ProductID, S.LocationID, S.TimeID$

Sales_Transaction			
ProductID	LocationID	TimeID	Sales
P1	L1	20/01/99	50
P1	L1	20/01/99	34
P1	L2	03/03/96	22
P2	L3	16/10/98	8
P2	L3	16/10/98	96
P2	L1	20/01/99	56
P2	L1	09/04/95	45
P3	L2	26/02/97	98
P3	L2	26/02/97	33

Table 2.1: The *Sales_Transaction* Relation

The benefit of the cube-by operator is that the user is no longer required explicitly to issue all the possible group-by statements. The user can now more conveniently navigate through the various levels of summary information in the database. The impact of the cube-by operator is greater when dimensions with multiple hierarchies are considered, the result of which may be equivalent to thousands of explicit group-bys.

2.2 The Set of Aggregations: A Hypercube Lattice

The aggregations derived from the relation in Table 2.1 can be organised into a lattice as illustrated in the direct acyclic graph (DAG) of Figure 2.1. [HRU96] introduced the lattice framework for OLAP to express the dependency between the queries (or aggregations) in the data cube. For example, if aggregation *A1* can be computed from aggregation *A2* then it can be said that *A1* is dependent on *A2*. In the example of the *Sales_Transaction* relation, *ProductID* can be computed from the *ProductID_LocationID*, hence, the *ProductID* is dependent on *ProductID_LocationID*. The cube lattice of the *Sales_Transaction* relation

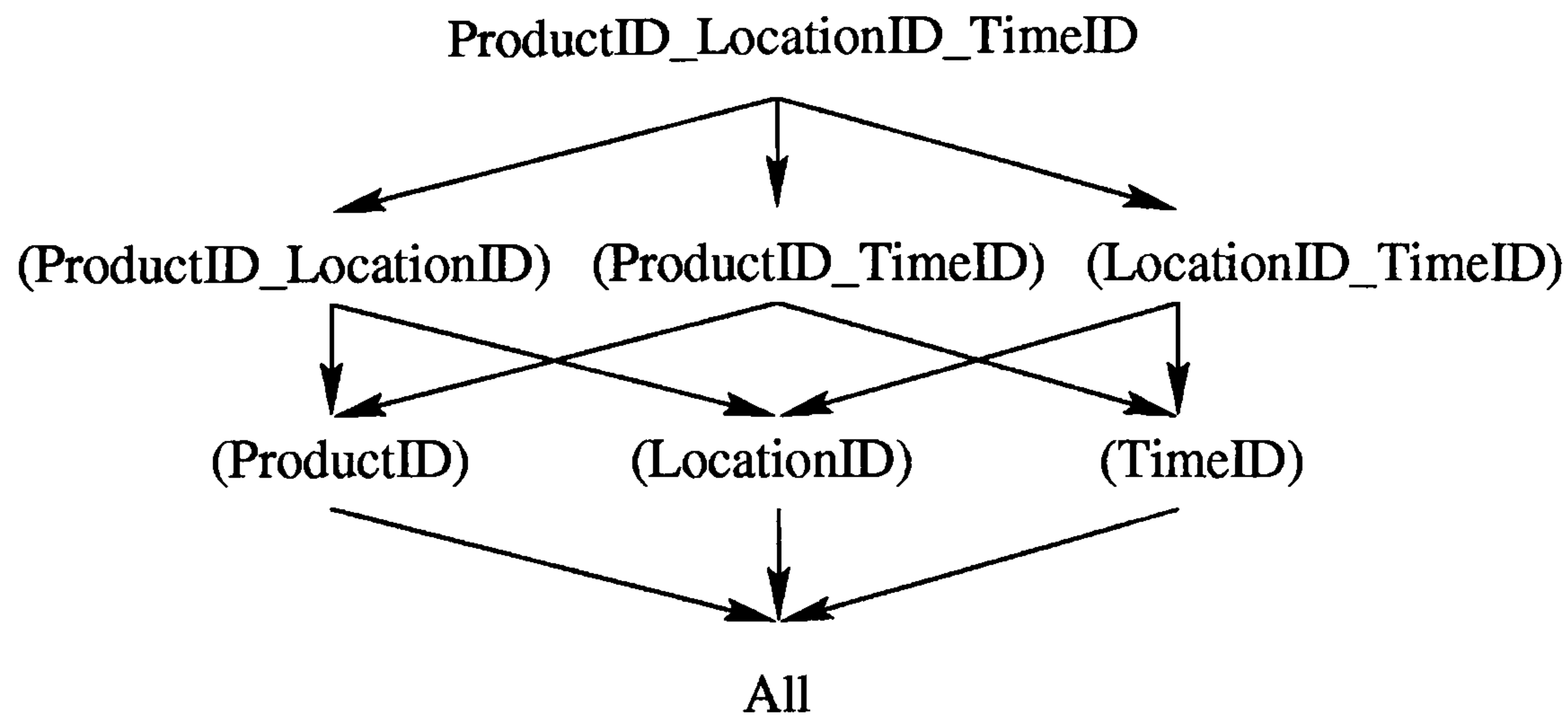

Figure 2.1: The Hypercube lattice

is shown in Figure 2.1. Following [KR82], a partially ordered set, where every finite subset has a least upper bound (lub) and greatest lower bound (glb), is called a lattice. The partial ordering of the queries is expressed by the $\preceq$ operator.

The Hypercube lattice is defined if the following criteria are preserved [HRU96]:

- There is a partial order $\preceq$ between the aggregate views in the lattice.
- There is a top view in the lattice and all views are dependent on the top view.

The ancestors and descendants of a lattice are defined as follows:

$$ancestor\ (a) = \{b \mid a \preceq b\}$$
$$descendant\ (a) = \{b \mid b \preceq a\}$$

equally for an aggregate a:

$$parent\ (a) = \{b \mid a \prec b,\ \nexists\ c,\ a \prec c,\ c \prec b\}$$
$$child\ (a) = \{b \mid b \prec a,\ \nexists\ c,\ b \prec c,\ c \prec a\}$$

where $a \prec b$ *means* $a \preceq b \wedge a \neq b$.

2.2.1 The Dimension Hierarchy Lattice

In Section 1.2.1 it was noted that the data in the dimension tables define *dimension hierarchies.* The hierarchies in the dimension tables can also be represented by a lattice. The *Product*, *Location* and *Time* dimensions could have lattices as shown in Figure 2.2. The bottom element 'none', means that there is no grouping by that dimension. Although

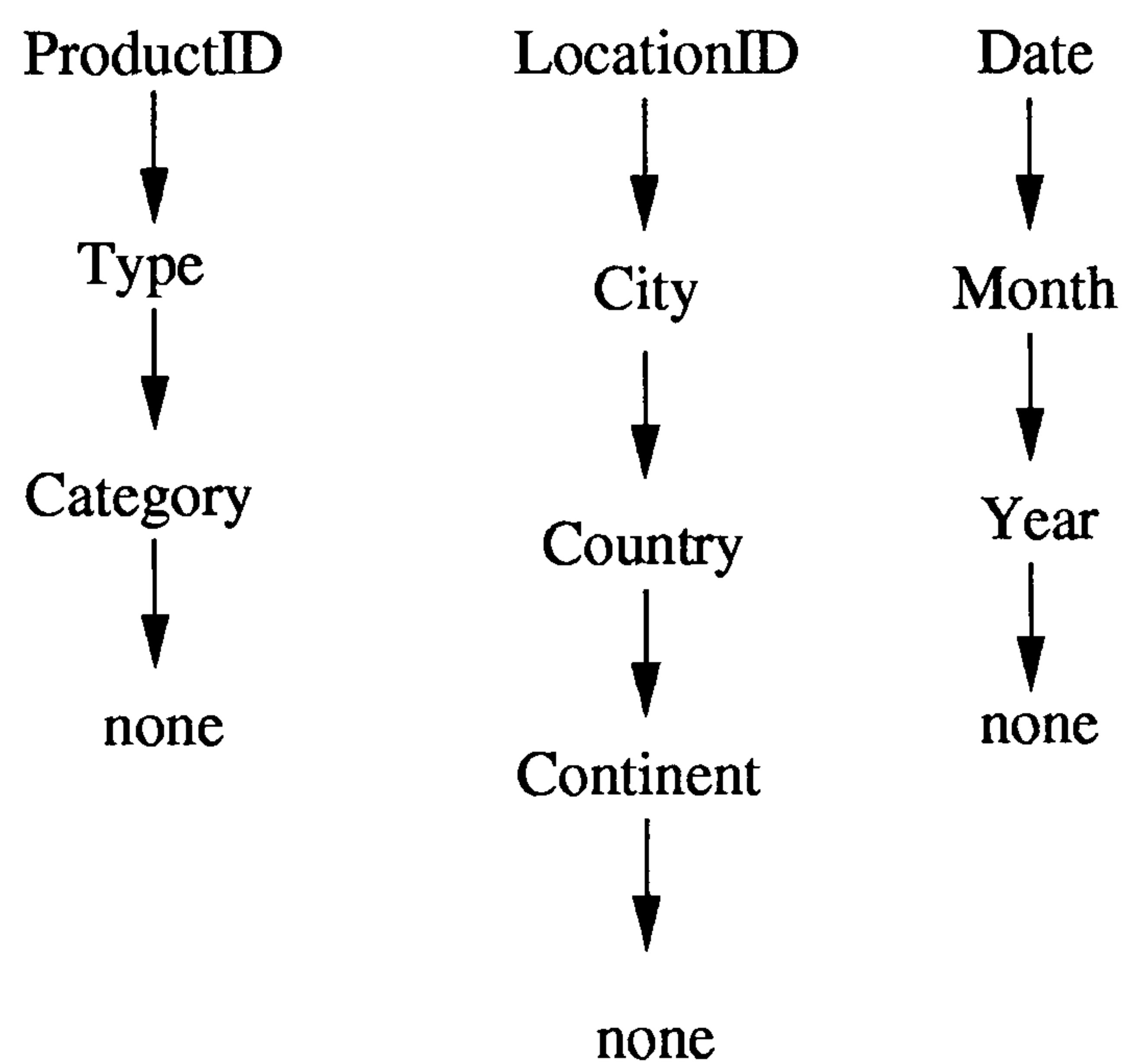

Figure 2.2: The dimensions hierarchy

the month and year in the *Time* hierarchy are comparable, some of the elements may not be directly comparable. For example, weeks and months do not strictly contain each other. A lattice representing the set of views - from the data cube and the hierarchies - can be obtained by grouping each combination of elements from the set of dimension hierarchies. The result is the direct product of the lattice for the fact table along with the lattices for the dimension hierarchies [HRU96]. Thus, instead of aggregating only a single value in the data cube, the various levels of each dimension hierarchy are aggregated.

2.3 The Role of Materialised Views in Data Warehousing

Before discussing the materialised views, the concept of a relational *view* will be discussed. A view is a virtual representation of a relation and defines a function from a set of base tables to a derived table. Every time the view is referenced, the function is recomputed. The existence of the views is significant because usually the actual schema of the database is normalised and the querying process is less effective when applied to the normalised relations (i.e., the join of dimension tables with the fact table) [GM99]. Thus to increase effectiveness, the views are defined as de-normalised relations. Consider a relation R with attributes a, b, c, d, e referred to as $R(a, b, c, d, e)$. The view $V1$ is defined as an aggregation of R in the grouping attributes $R(a, b, c)$ and can be expressed in SQL as follows:

CREATE VIEW $V1(V1.a, V1.b, V1.c)$ AS

SELECT $R.a, R.b, R.c$

FROM R

GROUP BY $R.a, R.b, R.c$

A view is called a *materialised view* when its tuples are stored explicitly in the database. The benefit of materialised views is that accessing the view is normally much faster than recomputing the view [Rous97]. The materialised view has the same characteristics as any data held in relational form and thus is like a copy of the data already in a form which can be accessed quickly. Materialised views also eliminate the need to expand and recompute the view definition each time the view is used [GM99]. Using the previous example, a new query on attributes a,b could utilise the materialised view $V1$. The benefit of using the view $V1$ instead of the base relation R is that $V1$ is already computed and is smaller in cardinality than R. The following SQL statement computes this aggregation in the grouping attributes *a,b* of the view *V1*:

SELECT $V1.a, V1.b$
FROM $V1$
GROUP BY $V1.a, V1.b$

Speed of access to information can be critical in a data warehouse environment where the query rate is high and the views are complex thus it is not feasible to recompute the view for every query. A view may also underlie many higher-level interfaces that are collectively queried at a frequency high enough to require this view to be materialised. To increase the efficiency of retrieval from views, index structures can be built on the materialised views [Rou82], [HRU96]. [GM99] and [SDJL96] have proposed algorithms to answer aggregation and group-by queries through materialised views. The process of updating a materialised view in response to changes in underlying data is called *view maintenance* [MQM97], [CKL+97].

2.3.1 Related Work

Earlier in Section 1.4 it was noted that the data cube can be implemented in three ways. In the first, no views are materialised and the aggregates are computed on-the-fly. In the second, everything is pre-computed and stored and in the third a subset of the aggregates is pre-computed and stored. However none of the previous techniques has made any systematic use of redundancy in multidimensional aggregates and there has been no work on the elimination of redundancy as a basis for the efficient selection and storage of materialised views. The following section will discuss the main methods for computation and selection of materialised views.

2.3.2 Methods for Aggregation

In the aggregation process, tuples belonging to the same grouping attributes must be brought together [Gra93]. There are three recognised methods for aggregation and these are based on nested loops, sorting and hashing. The analysis of general sorting and hashing algorithms is beyond the scope of this thesis so discussion will be restricted to their use in association with aggregation.

The Nested Loops method is the simplest, wherein the algorithm loops, for each input item, over a temporary file or array and accumulates this item. This method is not efficient for large inputs as the expected complexity is $O(N^2)$, where N is the number of tuples in a relation [Knu98].

In the sorting method, the goal is to bring equal items together so that aggregation in grouping attributes is easier. Sort-based aggregation is favoured in disk-based aggregation mostly because it does not require the output relation to fit into main memory. Expected complexity of the sorting is $O(NlogN)$ [Knu98].

In hashed-based aggregation, the grouping attributes are hashed and equal items can be found and aggregated when they are inserted into the hash table. If the entire hash

table fits into main memory, hash-based aggregation is easy to design and faster than sorting. The expected complexity of hashing is $O(N)$ [Knu98].

If the hash table does not fit into the main memory the table may be partitioned. Each partition requires a partial pass through the input relation. Novel methods of minimising the amount of data representing the aggregate views are presented in this thesis (refer to Chapter 3).

2.3.3 Computing the Data Cube

The goal in data cube implementation is to compute all 2^n possible aggregates as quickly as possible. To achieve fast computation, several optimisations are possible. For any given input base relation of arity k, there are k derivative aggregations of order $k-1$. Each of k aggregates can in turn be a parent of other derivative aggregates. However, as can be seen from Figure 2.1, each aggregation can be derived from several parents. [GBLP96] proposes an optimisation in which every group-by is computed from the smallest parent. For example, it is obviously faster to compute the aggregate AB from its parent ABC with size 1,000 records, compared to another parent ABD with 50,000 records. Other optimisations proposed by [AAD+96] and [SAG96] are: to cache results, amortise scans, share sorts and share partitions.

[GBLP96] proposed implementing the data cube using a main memory technique called the 2^N algorithm, where N is the number of dimensions. Providing the N-dimensional array can be fitted into the memory, the N-1 dimensional super-aggregates can be computed by projecting one dimension at a time. [GBLP96] also suggested that if the array is larger than the available memory, the cube must be organised by value, using sorting and hashing techniques and then computed by aggregating the organised data.

[SAG96] introduced the Pipe-sort and Pipe-hash algorithms. The Pipe-sort algorithm annotates each edge in the search lattice with two costs. The $A(e_{i,j})$ is the cost of

computing a child j from a parent i without sorting and $S(e_{i,j})$ is the cost of computing j from a parent i which needs sorting. In a graph representing the lattice, $(e_{i,j})$ is the edge between i and j. The algorithm proceeds level-by-level from the minimum to the maximum number of attributes. For each level k in the lattice, it finds the best way of computing level k from level $k+1$, thus reducing the problem to the weighted bipartite matching problem [SAG96]. Pipe-sort includes optimisations such as sharing sorting orders, smallest-parent, cache-results and amortized scans.

The Pipe-hash algorithm computes the group-by j from the smallest parent. The decision is based on size estimation techniques and the result is a minimum spanning tree (MST). When memory restrictions prohibit computing all the group-bys in the MST together, it has to be decided which group-bys should be computed together, when to allocate and de-allocate memory for different hash tables and which partitions to compute first. This problem is NP-complete and [SAG96] has proposed a heuristic which selects the largest subtree. The Pipe-hash algorithm includes various optimisations such as smallest parent, cache results, amortize scans and share partitions.

[DANR96] presents the Overlap method which takes advantage of any sorted aggregate and 'overlaps' the computation of different aggregates, reducing the number of sorting stages. Thus aggregates can be computed from a sorted parent in sorted order. Overlap minimises the number of scans needed using size estimation techniques to determine a plan for computing the aggregates. Thereafter, it sorts the base relation according to the order in which the rest of the group-bys will be computed. Several aggregates thus can be computed concurrently in the memory.

[ZDN97] presents an array based algorithm utilising 'chunks' of memory for efficient storage on disk. A chunk of an n-dimensional array is an n-dimensional sub-array which corresponds to a page. The array is stored in units of chunks so fragments of data are stored in memory at each processing time. This is similar to Overlap [DANR96] but provides

better memory utilisation for storing partitions. Although the algorithm is proposed for multidimensional data organisation, it is also suitable for relational structures.

[RS97] introduces a partition algorithm that divides the relation into small data cubes which are fitted into memory. Partition-Cube partitions the data on some attribute into small data units which fit into the memory. The algorithm breaks the relation into $n + 1$ smaller sub-cubes computations, n of which are likely to be smaller than the base relation. If there are T tuples in R then T/n tuples should be expected in each partition. The Memory-Cube algorithm is similar to Pipe-sort but performs better, determining the optimal set of paths needed to compute each group-by in the data cube. Potentially, this results in an optimal number of sorts.

2.3.4 Selection of Views for Materialisation

The selection of materialised views balances the trade-off between space and time in a more efficient way and is considered the most desirable. The selection of materialised views has been studied by [HRU96], [BPT97], [Gup97] and [SDN98]. These algorithms select a subset of aggregates for computation which is based on available disk space, the estimated size of the aggregate and the estimated benefit of pre-computing the aggregate. However, none of the existing techniques for the selection of materialised views has made any contribution to the elimination of redundancy in data representation.

[HRU96] considers a lattice with 2^n views and assigns each view a cost according to its size. The goal is to select which views to materialise so that the average query cost is minimised, given a fixed amount of space. The cost of the query is based on the 'linear cost model' in which the time to answer a query is taken to be equal to the space occupied by the view from which the query is answered. [HRU96] proposes a 'Greedy' algorithm which chooses to materialise a fixed number of views regardless of the space they use. After selecting a subset of views for materialisation, the benefit of a view is

computed by considering how this view can improve the cost of evaluating others, including itself. The view with the maximum benefit is selected for materialisation and this process continues until the fixed number of views have been selected. In the next step, the Greedy algorithm considers the problem of allocating a fixed amount of space instead of a fixed number of views. Here, the algorithm uses the benefit per unit space of an aggregate. Given the amount of space available for pre-computation and a set initially containing all aggregates in the lattice (except the raw data), the goal is to find the set of aggregates to be materialised. [HRU96] claimed that the benefit of the Greedy algorithm is at least 63% of the optimal case and also that the performance of the algorithm remains the same even when each view is unlikely to have the same probability of being requested in a query.

[SDN98] have evaluated the Greedy Algorithm [HRU96] and have shown that it needs a prohibitive amount of processing. Instead, the authors of [SDN98] proposed the PBS (Picked By Size) algorithm which picks aggregates for pre-computation in increasing order of their size. Given the amount of space available for pre-computation and a set initially containing all aggregates in the lattice, the goal is to find a set of aggregates to be materialised. [SDN98] claimed that the benefit of this algorithm is the same as the previous greedy algorithm [HRU96] but requires a fraction of the time. [SDN98] have noted that all aggregates equal in size to the database size have zero benefit. This means that any query which can be answered by scanning a view, can be answered at equal cost by scanning the raw data. PBS assumed that all aggregates have an equal probability of being queried. The authors of [SDN98] have also proposed a variation of PBS, the PBS-U, in which a user can assign probabilities to aggregates.

[BPT97] introduces the idea of user's response utilisation. If a set of user-specified relevant queries is available, exploiting this information may yield a significant reduction in resources (time and space). The authors of [BPT97] have also observed that the number of representative queries is extremely small in respect to the total number of elements of the

complete data-cube. Thus, the information about which queries are required is utilised to guide the selection of candidate views, i.e., which views if materialised, may yield a reduction in the total cost. However, note that this technique is application-oriented and requires a set of queries defined by previous requests.

Recently, [BR99] introduced the Iceberg-Cube as a reformulation of the data cube problem to selectively compute only those partitions that satisfy a user-specified aggregate condition defined by a selectivity predicator (HAVING clause). Thus the Iceberg-Cube problem is to compute only those group-by partitions with an aggregate value (e.g., count) above some minimum threshold. [BR99] compute the Iceberg-Cube in a bottom-up order by introducing the bottom-up-Cube algorithm (BUC). BUC builds the data cube by starting from a group-by on a single attribute, then a group-by on a pair of attributes and so on. Potentially BUC avoids the computation of large group-bys that do not satisfy the condition defined by the selectivity predicator (HAVING clause). The authors [BR99] claim that this optimisation improves computation by 40%. However, this approach is inherently limited by the level of aggregation specified by the selectivity predicate HAVING-COUNT(*).

2.3.5 Further Related Work

[RSC97] provides methods for partitioning the attributes in order to answer complex aggregate queries.

Compressed methods to reduce the data volume of the materialised views have also been proposed by [OG95], [WB98] and [KM99]. [OG95] presented an approach for joins between the fact tables and the dimension tables, based on the combination of join indices and bitmap indices. [WB98] introduced the Encoded Bitmap Indexing (EBI) as an optimisation of the simple bitmap indexing initially proposed by [O87]. The EBI, instead of storing n bitmap vectors (where n is the cardinality of an attribute) required by the

simple technique [O87], only requires $log_2 n$ bitmap vectors and a mapping table. EBI is proposed by [WB98] as a solution for attributes with large cardinalities in a data warehouse. [KM99] proposed a compressed architecture for the data warehouse environment. By distributing a dictionary across the users, the querying stage operates in a compressed representation. The updating of the views can also operate in the proposed compact form. As a result, faster computation time and lower storage cost compared to the uncompressed representation can be achieved.

Approximate methods have been proposed by [BS98] and [VW99]. [BS98] introduces the Quasi-Cubes as an alternative to the data cube. The proposed method is based on statistical models (linear regression) and stores multidimensional aggregates in a form which provides fast approximate answers. [VW99] method constructs the data cube through multiresolution wavelet decomposition. This method performs well in sparse multidimensional arrays.

Query optimisation techniques are also applied to the aggregate problem, e.g., [Sel88], [CS94], [YL95]. [Klug82] and [OOM87] present aggregation expressions, while aggregation processing in a data warehouse environment has been proposed by [GHQ95], [Wid95] and [LQA97].

Chapter 3

The Low Redundancy Concept

To allow users fast access to specified aggregate, the prevailing paradigm in OLAP has been for systems to pre-compute results and store them as materialised views. This requires a large amount of storage, which is justified on the traditional grounds that obtaining a result by accessing a table is faster than computing it - a speed for space trade-off.

This thesis introduces the L-R approach as a novel alternative paradigm for OLAP. The L-R identifies and eliminates redundancy in multidimensional aggregates. The approach is based on the fundamental observation that if redundant data can be identified then it need neither be processed nor stored.

The two key elements of the L-R approach are as follows:

1. Many of the possible aggregates are directly derivable from their parent input relation without any processing. These redundant aggregates provide little, if any, benefit to the user and will be referred as *Totally-Redundant* aggregates. New theory proposed here, derived from relational theory, provides a means of determining by inspection which views belong to this category. The practical implication of this are that a large percentage of views require neither processing nor storage (e.g., 70% – 85% in TPC-D 60K, with 10 dimensions).

2. In the remaining aggregates, a further redundancy occurs when a subset of the tuples of the aggregate relation is directly derivable from a subset of its parent relation, in which case only those tuples which are different from the parent need be stored. These aggregates will be referred to as *Partially-Redundant* views. The differential representation requires only a fraction of the space compared to that required by conventional storage of materialised views (e.g., approximately 27 times less space is required in TPC-D 60K with 10 dimensions).

Table 3.1 introduces notation which will be used in later discussion.

Notation	Description
R	Input (or parent relation of R')
R'	Aggregation (or child relation of R)
t_R	Tuple in R
$t_{R'}$	Tuple in R'
C_R	Cardinality of R
$C_{R'}$	Cardinality of R'
S_t	Set of grouping attributes in t_R
$S_{t'}$	Set of grouping attributes in $t_{R'}$
M_R	Measure of interest in R
$M_{R'}$	Measure of interest in R'

Table 3.1: Notation of main components

3.1 Totally-Redundant Views

The *candidate keys* are classified into two types: *Definitional* and *Observational keys.*

Definition 3.1.1 *Definitional Keys are those keys which are defined as part of the database schema (e.g., by the database designer).*

Definition 3.1.2 *Observational Keys are those keys which are defined empirically.*

Thus an Observational key is invariant in a read-only database but may be modified by updates to the dataset. A Definitional key always possess a unique identification property despite updates.

Definition 3.1.3 *A tuple $t_{R'}$ is defined to be* **group-by - equivalent** *or* **g-equivalent** *($\Rrightarrow$) to a tuple t_R if, and only if, the set of grouping attributes $S_{t'}$ is a subset of the set of grouping attributes S_t and the measure of interest is equal for both R' and R (see Figure 3.1).*

$$t_{R'} \Rrightarrow t_R \text{ iff } S_{t'} \subset S_t \quad \textbf{and} \quad M_{R'} = M_R$$

(P2,L1,T2,30) ⟷ *(P2,L1,30)*

Figure 3.1: The g-equivalent tuple

Definition 3.1.4 *A relation R' is defined to be* **g-equivalent** *($\Rrightarrow$) to a relation R if, and only if, for every tuple in R' there is a g-equivalent corresponding tuple in R and both relations have the same cardinality.*

$$R \Rrightarrow R' \quad \text{iff} \quad (\forall\, t_R \;\exists\, t_{R'} \quad \textbf{such that} \quad t_{R'} \Rrightarrow t_R) \quad \textbf{and} \quad C_{R'} = C_R$$

3.1.1 Extending the Relational Theory: Totally-Redundant Views

Theorem 3.1 *When the result relation R' of an aggregation has the same cardinality as the parent relation R then each tuple $t_{R'}$ is g-equivalent to the corresponding tuple t_R, both in its grouping attributes and in its measure of interest.*

Proof In an aggregation operation each tuple $t_{R'}$ is derived either from a single tuple or from several tuples of the input relation R (refer to Section 3.1). If a tuple $t_{R'}$ is a result of several tuples in R then there is a reduction in cardinality of the relation R' relative to the relation R. Thus, if the cardinality of R and R' is the same, then each tuple $t_{R'}$ must have been derived from only a single tuple t_R, and hence must be g-equivalent to the corresponding tuple of R in both its projected dimensional values and its measure of interest.

Theorem 3.2 *Any aggregation of a relation R over any set of domains which includes a Candidate Key, produces a result relation R' in which each resulting tuple must be g-equivalent to the corresponding tuple of R in both its grouping attributes and its measure of interest.*

Proof Each candidate key of a relation R has the property of uniquely identifying each tuple of that relation. Any projection or aggregation of R that includes a candidate key preserves the same number of tuples. Thus, any aggregation or projection which includes a candidate key of R, produces a result relation R' with the same cardinality as R. Thus, (by Theorem 3.1) each tuple in R' must be identical to the corresponding tuple of R in both its projected dimensional values and its measure of interest.

Theorem 3.3 *(Converse of Theorem 1): When an aggregation or projection of a parent relation R over a set of domains produces a result relation R' with the same cardinality as in R, then that set of domains contains an Observational candidate Key of both R and R'.*

Proof From the theory described in Section 1.1, a domain (or combination of domains) of a given relation, whose values uniquely identify each element (n-tuple) of that relation, is called a *candidate key*. If an aggregation or projection of a parent relation R produces

a resulting relation R' with the same cardinality, then the dimensions over which the aggregation has been carried out must uniquely distinguish each tuple of R. Each resulting tuple must have been derived from a single parent tuple (if this were not so, then some aggregations from several tuples would have occurred, with a resultant reduction in the cardinality of R'). Thus, if the cardinalities of R' and R are the same, the dimensions of the aggregation must include a candidate key of R.

3.1.2 Example of Totally-Redundant Views

Table 3.2 shows the input relation R and Table 3.3 shows the g-equivalent aggregate relation R'. R and R' have equal cardinality and for every tuple in R' there is a g-equivalent corresponding tuple in R. The aggregate relation R' is redundant since it can always be produced by a simple [1] projection of R.

3.2 Partially-Redundant Views

Definition 3.2.1 *A relation R' is defined to be* **similar** *to a relation R if the fraction $C_{R'}/C_R$ of g-equivalent tuples in R' is within a threshold t_s.*

$$\textbf{i.e.}\quad t_s \leq C_{R'}/C_R \prec 1$$

The value of the threshold t_s is a dynamic variable which can be defined by the database designer.

[1] Simple projection refers to a projection which does not require duplicate elimination

R			
Product	Location	Time	$Total_Sales$
P1	L1	T1	80
P2	L3	T4	20
P1	L2	T3	50
P4	L1	T1	30
P3	L1	T3	80
P4	L3	T2	100
P1	L3	T1	45
P3	L2	T3	70
P2	L1	T2	30

Table 3.2: The Input Relation R

R'		
Product	Location	$Total_Sales$
P1	L1	80
P2	L3	20
P1	L2	50
P4	L1	30
P3	L1	80
P4	L3	100
P1	L3	45
P3	L2	70
P2	L1	30

Table 3.3: The g-equivalent aggregate relation R' of relation R (Table 3.2)

3.2.1 Extending the Relational Theory: Partially-Redundant Views

Theorem 3.4 *An aggregate relation R' is* **Partially-Redundant** *with regard to the parent relation R, if a subset of tuples in R' are g-equivalent to those in R.*

Proof Since R' has fewer tuples than R, some of the tuples in R' must be aggregates of several tuples in R, while the remaining tuples in R' must be directly g-equivalent (simple projection) to the corresponding individual tuples present in R.

Partially-Redundant views may be represented by means of the union of two relations. The first relation, is the set of tuples in which each tuple is an aggregate of several tuples in the input relation. The second relation is the set of tuples which are g-equivalent to and thus can be derived from the parent relation.

By storing only the first relation significant changes can be achieved. This relation is called the *Difference* or *Delta* relation. In the querying stage the aggregation can be reconstituted from its Difference representation as will be discussed in Section 3.3.

3.2.2 Example of Partially-Redundant Views

Figure 3.2 shows the relation *R(p,s,t)* and its aggregation *Rc(p,s)*. In Rc two tuples *(P2, S2, 70)* and *(P1, S2, 20)* are g-equivalent to the corresponding tuples in *R (P2, S2, T1, 70)* and *(P1, S2, T2, 20)* respectively. The remainder is the Difference or Delta relation *(Rd)*, which is an aggregate of three tuples in R represented by R_α.

The two relations R and R_c are presented indirectly. R_c can be partitioned into two relations R_d and R'_t. The tuples of R'_t are g-equivalent to the corresponding tuples R_t of R and thus redundant. This redundancy in the data representation of R_c is avoided by storing only R_d, assuming that the parent relation is stored. The R_d relation consists of the tuples of R_c which are not g-equivalent to any tuples in the parent relation R.

3.3 Difference Algebraic Equations

The realisation of the Partially-Redundant view approach implies two-stages. In the first stage (the computation stage), the Difference relation is extracted from the aggregation

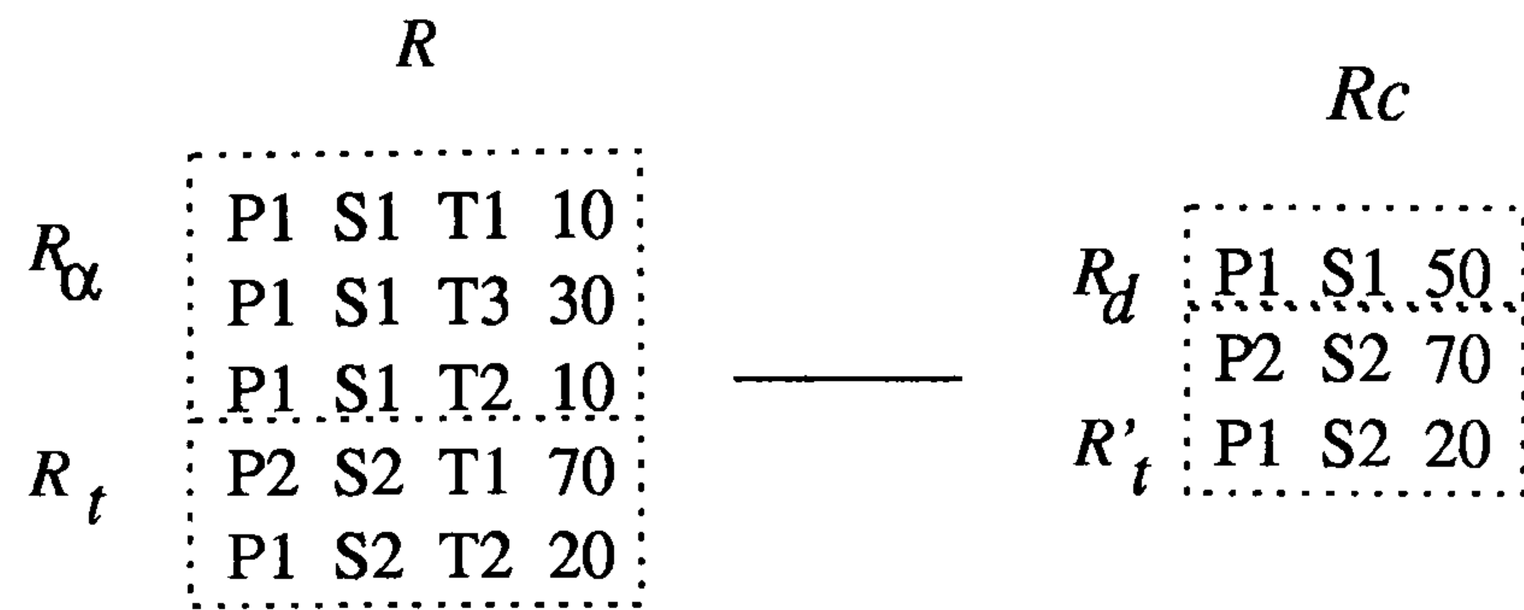

Figure 3.2: Producing the Difference representation

using the Aggregation Difference (AD) algebraic equation. In the second stage (the retrieval stage), the aggregate is reconstructed from its Difference form using the Aggregation Reconstruction (AR) algebraic equation. These stages are similar to the well-known computation of multidimensional aggregates and their later access from the materialised views.

The AD Equation

The AD equation would require the following operations:

$$R'_t = R_c \rtimes R$$

$$R_d = R_c - R'_t$$

where ($\rtimes$) is the semi-join [2] operator in the grouping attributes and ($-$) is the difference operator.

[2]Given two relation R_1 and R_2, a natural join followed by a projection on the first operand is a *semijoin* written $R_1 \rtimes R_2$. So $R_1 \rtimes R_2 = \pi(R_1 * R_2)$. Because only R_1 attributes enter into the answer relation, the purpose of R_2 is simply to reduce R_1 to those tuples for which the common attribute values also appear in R_2 [John97].

The AR Equation

The AR equation reconstructs the aggregation relation R_c from its Difference R_d in the following operations:

$$R'_t = (R_d \overline{\rhd\!\!\lhd} R)$$

$$R_c = R_d \cup R'_t$$

where ($\overline{\rhd\!\!\lhd}$) is the anti-semi-join [3] operator in the grouping attributes and ($\cup$) is the union operator.
The above equations show that the difference representation can, in principle, be implemented in any relational system.

3.4 Implementation Considerations

Totally-Redundant views can be identified by the Key-algorithm which reduces the problem of redundancy by finding the set of Observational keys in the base relation. The detection of Observational keys is accomplished by checking for duplicity of tuples existing in some group-bys. In this sense, the algorithm might appear to be simple or slow. On the contrary however, the algorithm is very fast due to its bottom-up order; the Key algorithm starts from the bottom (or the small group-bys) and moves to the top of the lattice and examines - by scanning for duplicates - if the particular group-by schema is a key of the base relation. If the algorithm identifies a key, this key will then automatically eliminate from further examination all those group-bys with schema, including the key schema. This process follows an exponential reduction of group-bys for examination and

[3] The anti-semijoin is a semijoin with an inequality (<>) predicate.

is responsible for the fast performance of the Key-algorithm. The benefits derived from the bottom-up order dictates that the Key-algorithm operates separately to the data-cube computation and not simultaneously. Section 3.6.1 describes the complexity of the Key algorithm in more depth.

Totally-Redundant views can be identified in relations other than the base input relation. A recursive version of the Key-algorithm can be applied to examine the derivative aggregations for keys. This method provides an optimised solution especially when there are no Observational keys in the base input relation.

The equations given earlier present the realisation of Partially-Redundant views as an extraction of the Difference representation (aggregate tuples) from the already aggregated relation. The algorithm proposed in this thesis (refer to Section 3.6) adopts a different route, whereby the separation between the aggregate and the non-aggregate tuples takes place during the aggregation operation and not after it.

3.4.1 View Types

The types of views used to handle the different data representations are:

- *The View* as defined in section 2.2. This view is utilised to represent the virtual representation of a relation. It includes a pointer to the parent input relation and may have the means to select the tuples and domains from the parent input relation. The View in the L-R approach represents a Totally-Redundant view.
- *The Stored-Relation* is the conventional materialised view which is used to store either the whole aggregation or the Difference representation.
- *The Semi-Stored View* is the view which represents a union of a Stored-Relation and a View.

The View and Semi-Stored View can be applied recursively. For example, it is possible to construct a View of a View, or a View of a Semi-Stored View as well as a View of a Stored-Relation. To avoid dealing with different levels of interpretations, the preferred implementation path is to materialise any View which is a parent of another. The cost of this materialisation is discussed in section 4.2.3. and shown in Figure 4.7. The different types of views are shown in Figure 3.3.

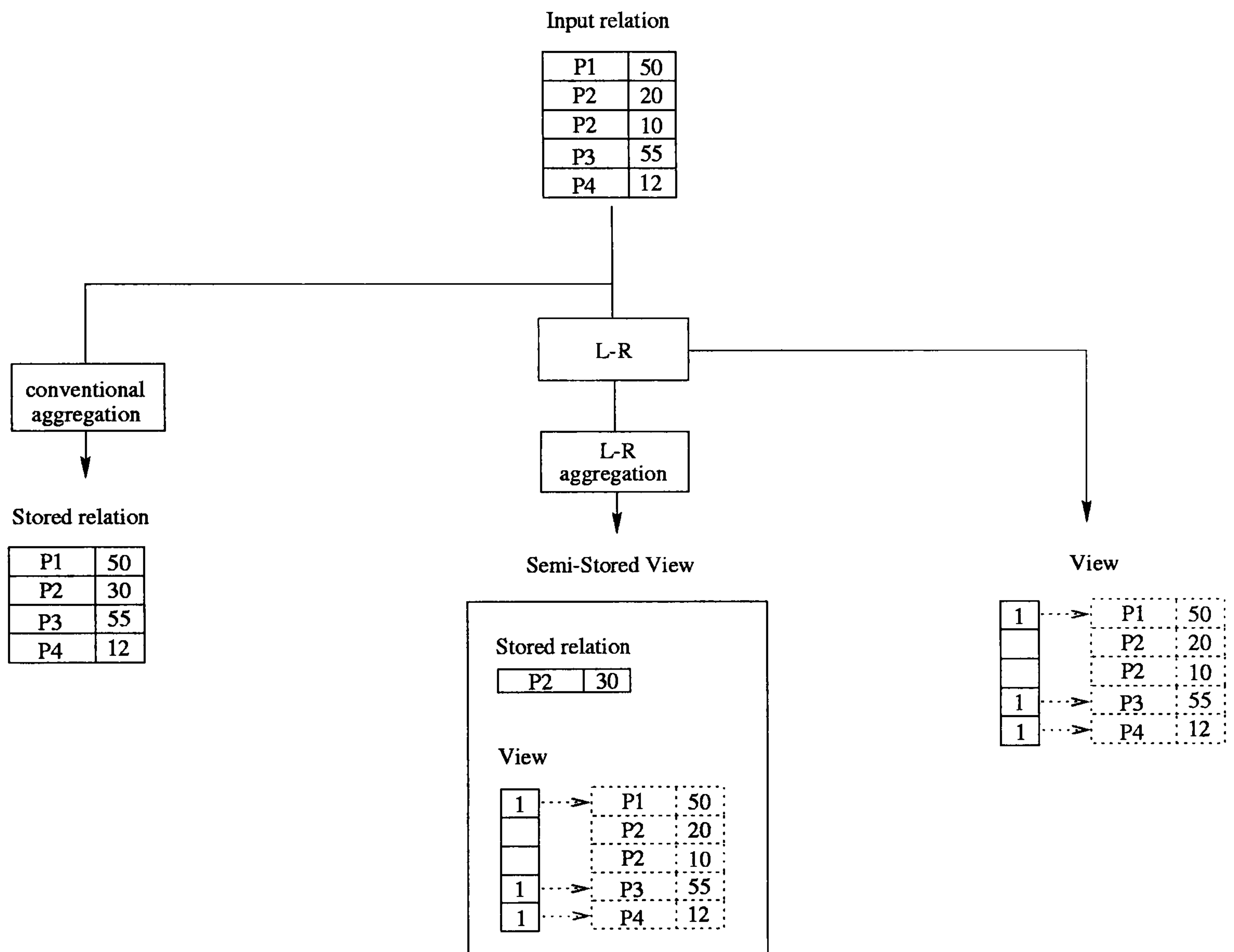

Figure 3.3: The types of views in L-R

3.5 Implementing the Totally-Redundant views

A system can utilise Theorem 3.1 by avoiding the explicit computation and separate storage of any aggregate which includes a key. The original relation may then be treated as containing the aggregate's virtual representation. This not only has a major impact on the computation time, but it also reduces the storage requirements of the materialised views. The approach proceeds in two stages:

- **Stage 1:** Determination of the set of Observational keys.
- **Stage 2:** Computation of the data cube by utilising the set of Observational keys (found in stage 1). This excludes the processing and storage of Totally-Redundant views.

3.5.1 The Key Algorithm

Briefly stated, the approach adopted empirically determines all the Observational keys [4] present in a given parent relation prior to the materialisation of the data cube. The algorithm examines all possible aggregates in the data cube and classifies each either as Totally-Redundant or not.

Thus, the algorithm examines whether each group-by includes one of the already detected keys - if so it can be categorised immediately as g-equivalent to the input relation and hence Totally-Redundant. The remaining group-by, with maximum size smaller than the size of the input relation, can not be candidates for the equivalence property. Potential aggregates which are not in either of the above categories are tested - to see whether any two tuples of the input relation are combined during aggregation - using the

[4]The candidate key can be determined either by definition (database catalogs) or using the [LO78] algorithm to find keys for a given set of attributes' names and a given set of functional dependencies. However, this approach would require knowledge of the functional dependencies. Therefore the *Key Algorithm* is proposed for the identification of Observational keys in the data cube.

First Duplicate Detector (FDD) routine described later in this section.

The Key algorithm proceeds from the minimum (level K-2 in Figure 3.4) to the maximum arity (level K) and uses the FDD to scan each group-by until it detects the first duplicate tuple. When a duplicate is found, the current group-by is an aggregate (not g-equivalent relation) and hence, according to theorem 3.1, not a key.

If there are no duplicates, then the schema of the group-by is an Observational Key. With the discovery of a key, the algorithm eliminates from further consideration all subsequent group-bys of greater arity including the key in their schemas. Such group-bys are thus also Totally-Redundant views.

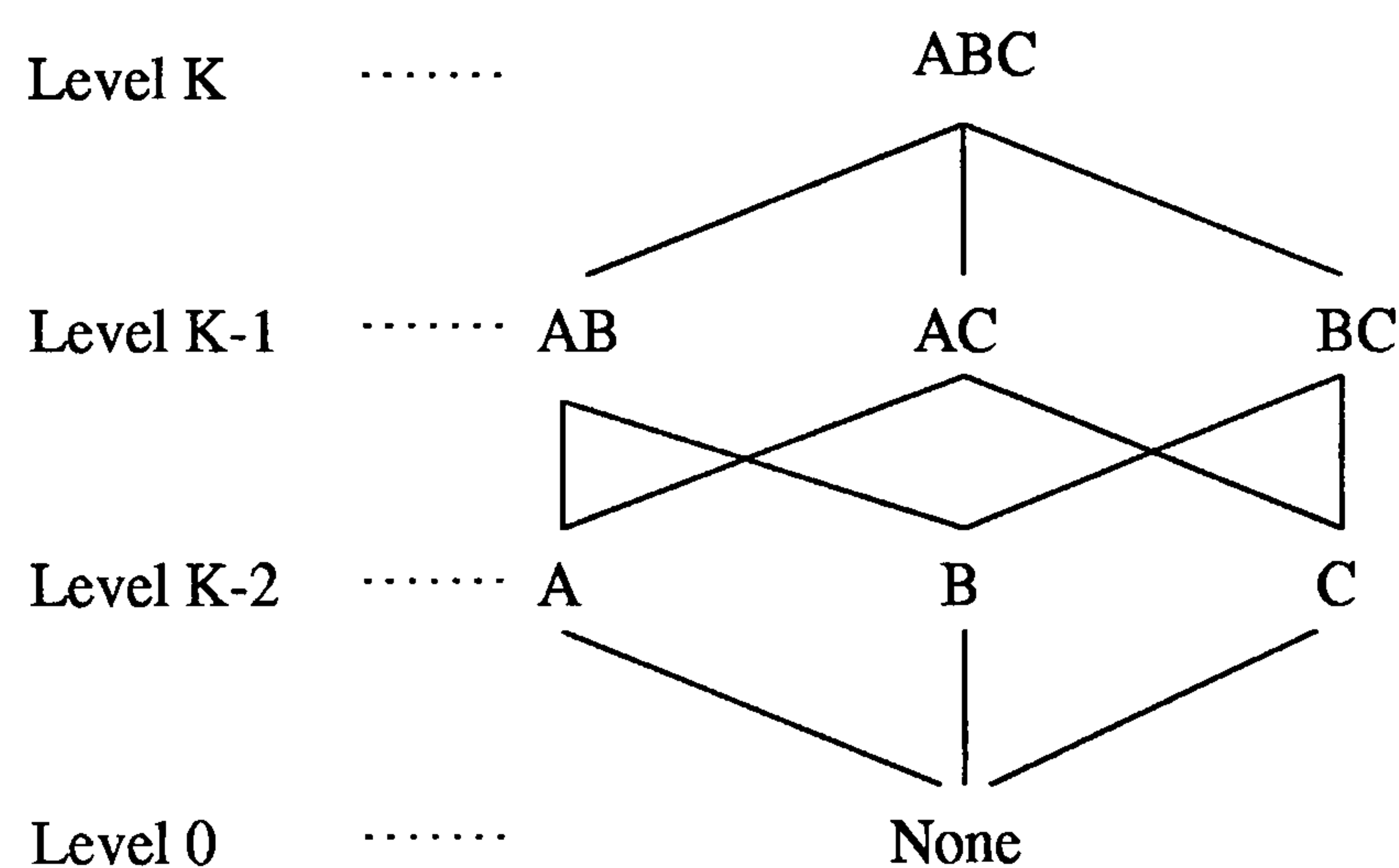

Figure 3.4: The cube lattice

3.5.2 The Complexity and Performance of the Key Algorithm

Given a relation $R(A1, A2, ..., An)$ with n dimensions, the complexity of the algorithm is $O(C * 2^n)$, where n is the number of dimension attributes and C is the cardinality of each group-by. C is an upper bound since the Key algorithm exits at the first duplicate which is normally detected before completing the scan of the whole group-by. The supersets of each Observational key must also possess the unique identification property and hence

these group-bys too can also be eliminated by inspection. If a schema of m attributes has been recognized as an Observational key of an *n-dimension* relation R, then the number of times the schema's m attributes will appear in the 2^n possible aggregations is 2^{n-m}. Thus, the smaller m, the greater the savings. The maximum benefit which can be derived occurs when $m = 1$ and the least benefit when $m = n$ and thus there is no key (n is the superset in R). This provides very significant leverage in the algorithm. For example a data cube of ten-dimensions would produce 1,024 aggregates. A key of two dimensions would reject $2^{10-2} = 2^8 = 256$ aggregates as Totally-Redundant and thereby no computation or storage would be required for them.

To further reduce the number of candidate keys, a group-by is not considered if the upper bound of its cardinality is smaller than that of the input relation. The proposed method to identify the upper-bound is the computed product of the dimension cardinalities.

The performance of the Key-algorithm is very efficient as it requires only approximately 10% of the conventional cube time. Figure 3.5 illustrates the time taken to compute a complete data-cube of 4, 5, 6, and 7 dimensions for the TPC-D [TPC98] dataset in the scale factor 0.1 (600K tuples).

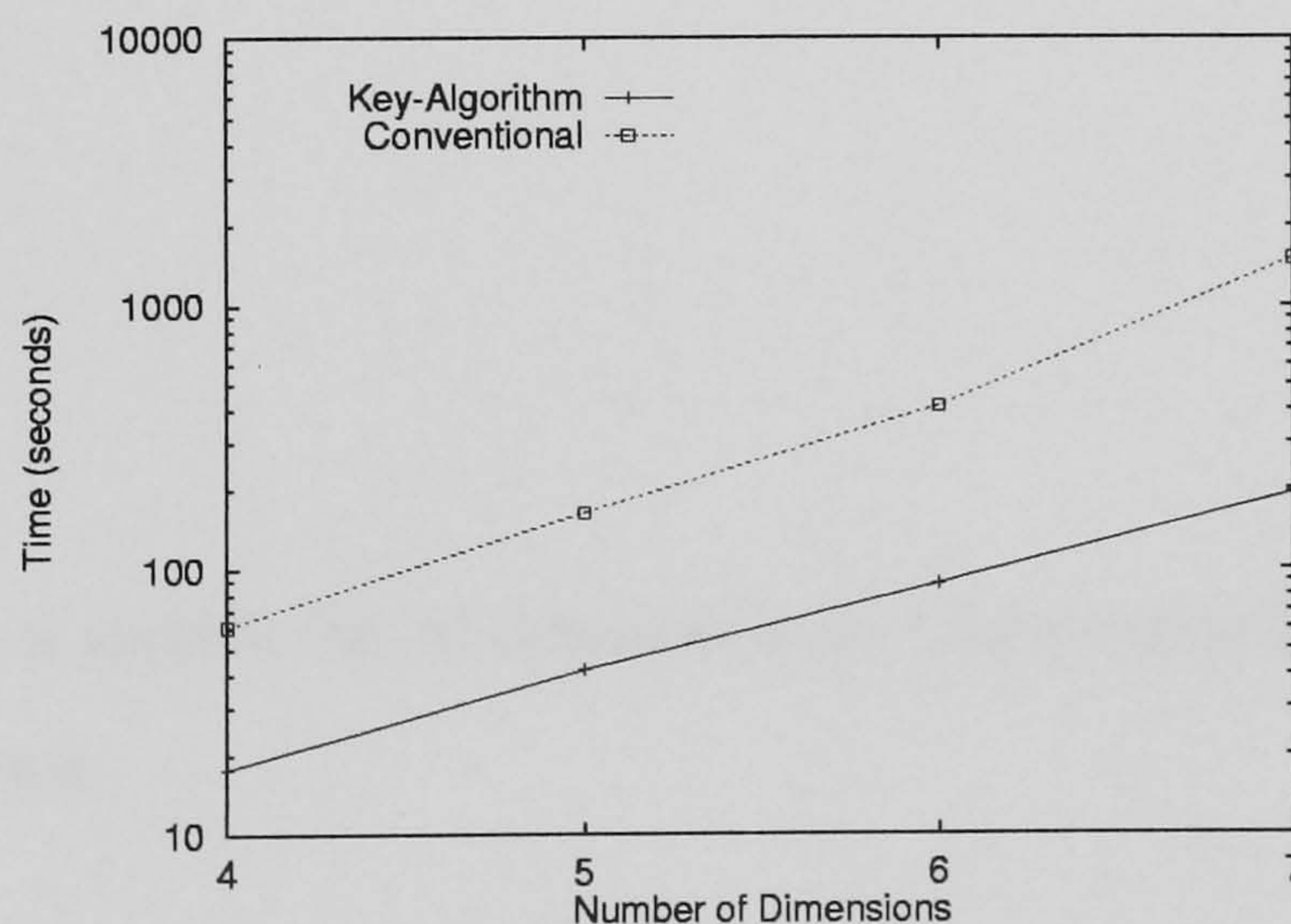

Figure 3.5: The performance of the Key algorithm

The Key algorithm

Input: search lattice of the input Relation R

Output: Set of Observational Keys K - array of strings

```
i := 0;
s := 0;
K := null;
while  i < NoOfCombinations - 1 do  (* NoOfCombinations = 2^n *)
   if GroupBy[i].size < R.size then begin   (* The size is an upper bound *)
     if GroupBy[i].schema ∈ K(s) then   (* This is a redundant GroupBy *)
      i := i + 1;
      else if found duplicate then  (* First Duplicate Detector *)
      i := i + 1;  (* This GroupBy is an aggregated relation - it's schema is not a Key *)
      else begin
                  s := s + 1;
                  add the GroupBy schema to K(s)
      end;
   end;
i := i + 1;
return set K;
end;
```

To examine whether a specific set of domains is an Observation key the First Duplicate Detector (FDD) is used.

The First Duplicate Detector

This is a simplified hash-based aggregation which exits true when a duplicate is found, false if none is detected. The complexity of the first duplicate detector is C, where C is the cardinality of group-by. In practice however, the average complexity is much smaller since a duplicate tuple is usually found (if it exists) before the full scan of the group-by. The FDD is closely related to theorem 3.1.

3.5.3 The Recursive Key Algorithm

The Key-algorithm, as described earlier, identifies Totally-Redundant views on the basis that they are g-equivalent to the input base relation. Further redundancy can be eliminated by applying the Key-algorithm recursively to the derivative aggregate views. The experiments outlined in Chapter 4 indicate that a further reduction in storage of up to 60% can be achieved when redundancy of the derivative relations is eliminated. Overall, Totally-Redundant views effect storage savings of up to 85% (TPC-D 60K with 10 dimensions) of the volume.

3.6 Implementing the Partially-Redundant Views

In implementing the Partially-Redundant views the goal is to extract the Difference tuples from the aggregate relation (refer to Figure 3.2).

Each Different view has an associated schema denoting the grouping attributes and a bit-array where a bit is set to one, corresponding to each tuple in the parent relation which is also found in the aggregate. The next section describes two algorithms for fast implementation of the Partially-Redundant views. The task of the proposed algorithms is to filter out the aggregated tuples (or Differences) from the aggregate relation and also identify the g-equivalent ones. These algorithms have been used to demonstrate the feasibility of the L-R methods and although are not proposed as the optimum solution, they have successfully demonstrated the effect of the L-R approach.

3.6.1 The Aggregation Algorithms

The first algorithm, called the Bit-array aggregator or B-aggregator, utilises bit-arrays. The B-aggregator has been used to evaluate the L-R experimentally. The method is similar to that of [Bloom70], [MTD76], [SL76] and [Babb79] and is described in Appendix D.

The second algorithm, called the Vector aggregator (or V-aggregator), is an improvement of the B-aggregator algorithm. Section 3.6.2 describes how the V-aggregator operates more efficiently in extracting the Differences. This is achieved by introducing a dual-resolution vector which filters the aggregated tuples from the g-equivalent ones while aggregating.

The innovation in both algorithms is that the extraction of the Differences occurs during aggregation and not after it.

3.6.2 The B-Aggregator

The B-aggregator algorithm is a hash-based aggregation in which a hash function is applied to the tuples, defined by the grouping attributes, in the parent relation. During the first pass over the input relation, hash values generated by tuples are entered into the first bit-array B_1. Tuples which generate a hash value already present in B_1, enter their hash value into a second bit-array B_2. Thus the hashed values entered into B_2 indicate candidate aggregated tuples of the input relation.

Collisions may be caused by the hashing function. Therefore the algorithm performs a second scan of the input relation to ensures that the tuples are 'real' aggregate tuples. The scan achieves this by searching for tuples whose values are in B_2 and compares these tuples with a table containing the 'real' aggregates. After this scan, any aggregate tuple found to be derived from a single tuple of the input relation is transferred to the set of g-equivalent tuples. This algorithm is illustrated in Figure 3.6.

The result of the aggregation is two sets of tuples, the set of the 'real' aggregates and the set of g-equivalent tuples contained in the parent relation. Both are represented explicitly as a Semi-Stored view which behaves as the union of the two components (see section 3.4.1).

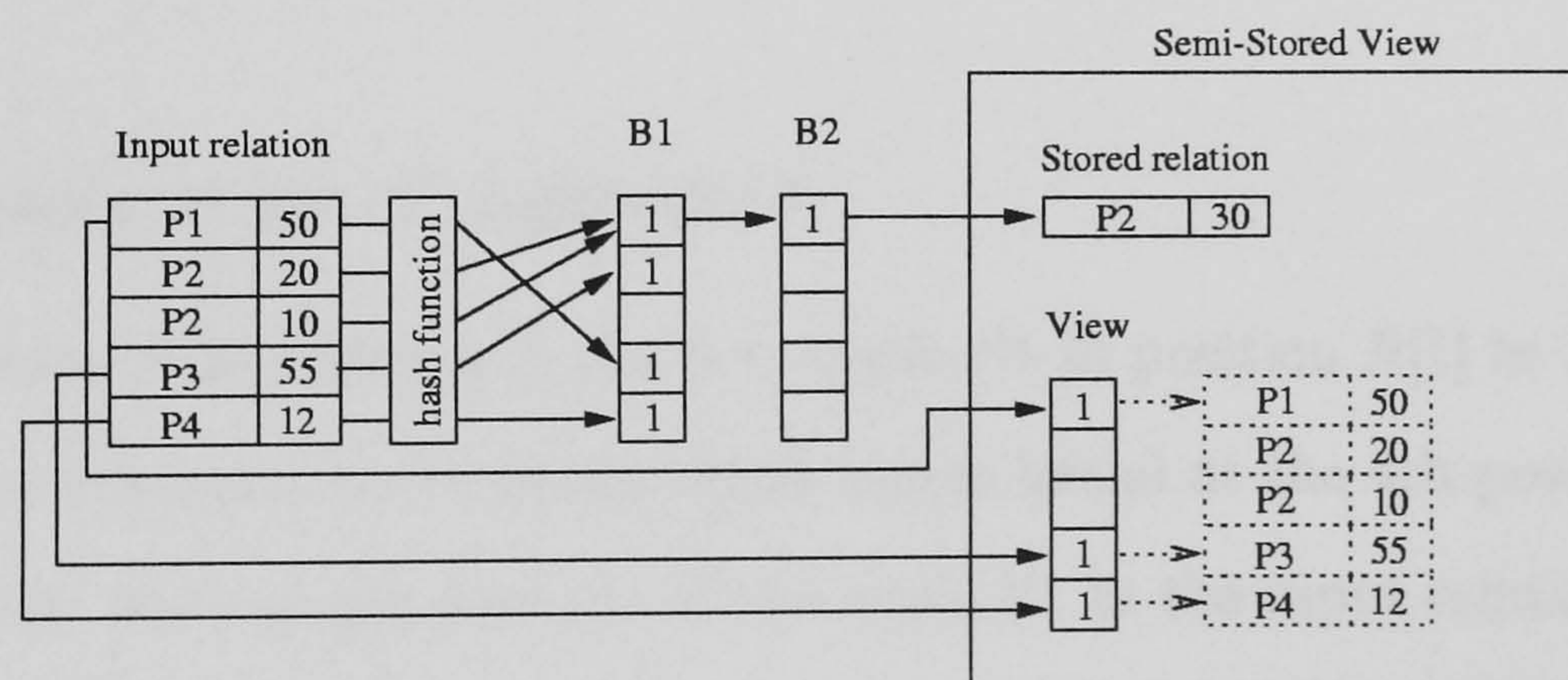

Figure 3.6: Performing an aggregation using the B-Aggregator

3.6.3 The V-Aggregator

The vector or V-algorithm is also a hash-based aggregation but one which accomplishes the aggregation in a single pass of the input relation. Initially, a hash function is applied to the tuples, defined by the grouping attributes, in the parent relation. The tuple hash value is used as the offset to a vector and this offset position contains the tuple number.

At the first tuple, a hash value is encountered and the corresponding entry in the vector will be in its initial value (0) and is assigned the negative sign ($-$). A negative entry in the vector corresponds to a tuple in the input relation. When a tuple hashes to a non-zero entry in the vector and the value is negative, the corresponding tuple is then copied from the input relation to the stored relation (The Difference R_d) and the entry is now switched to positive to give the tuple number in R_d. If, however, the vector's entry in the vector is already positive, the new tuple is aggregated with the corresponding tuple in R_d. The entries of the vector are assigned accordingly:

($-$) *for g-equivalent (or non-aggregate) tuples allocated in* R

($+$) *for Difference (or aggregate) tuples allocated in* R_d

3.6.4 Example of the V-Aggregator

Given the relation R in Figure 3.7, the first tuple $P1$ in position $R[1]$ in R is hashed and its hashed value is stored in the vector (hash access table) at the 4th position. The entry (-1) in the vector denotes the position of the tuple P1 in the input relation R. Similarly, tuple P2 is hashed in position 1 with the entry (-2) (Figure 3.7(a)). When an aggregation occurs, e.g., the 3rd tuple P2, a re-arrangement occurs. The entry (-2) of the tuple P1 in the vector is changed to point to the new location of the tuple P2 in relation R_d (i.e.,

$R_d[1]=1$), so the first entry of the vector becomes (+1) (Figure 3.7(b)). After the re-arrangement the second tuple P2 will aggregate with the tuple in position 1 in the table R_d (Figure 3.7(c)). In sequence, tuple $P3$ is then hashed in position 2 with the entry (-4).

When a collision occurs, a two step procedure takes place. In this example (see Figure 3.7(c)), tuple P4 collides with the tuple P1 in the hashed entry 4, (-1). The first step is for the tuple P1 to be transferred to the table R_d in the next available position ($R_d[2]$) and the 4th entry in the vector becomes (2), denoting the 2nd position in table R_d. In the second step, tuple P4 is ready to be allocated to the next available entry in the hash access table, thus it allocates the 5th position to the entry (-5) (Figure 3.7(d)). The appropriate changes are assigned to R_d and the single access table. Every time a tuple is aggregated the single table entry becomes 0.

Finally, the single table is examined and the set of tuples with zero entries is thus the Difference relation which is stored in a Stored-Relation (R_d). This set has been separated from the set of g-equivalent tuples, which is stored in a View (bit-array or $R - R_d$). The final aggregate relation R_c is the union of the R_d (Stored-Relation) and $R - R_d$ (View) (in line with equation AD in Section 3.2).

3.7 Computing the L-R Data Cube

The Implementation of the optimised cube is performed in two stages.

During the first stage, aggregations which are Totally-Redundant are identified and each is represented by a view structure. This process is carried-out by the Key Algorithm which constructs a key list, as was described in section 3.6. The output includes the set of keys and the set of Totally-Redundant views, each represented by its view structure.

In the second stage, the data cube is computed according to the traditional 2^n combinations. However, now the algorithm utilises the information captured at the first stage,

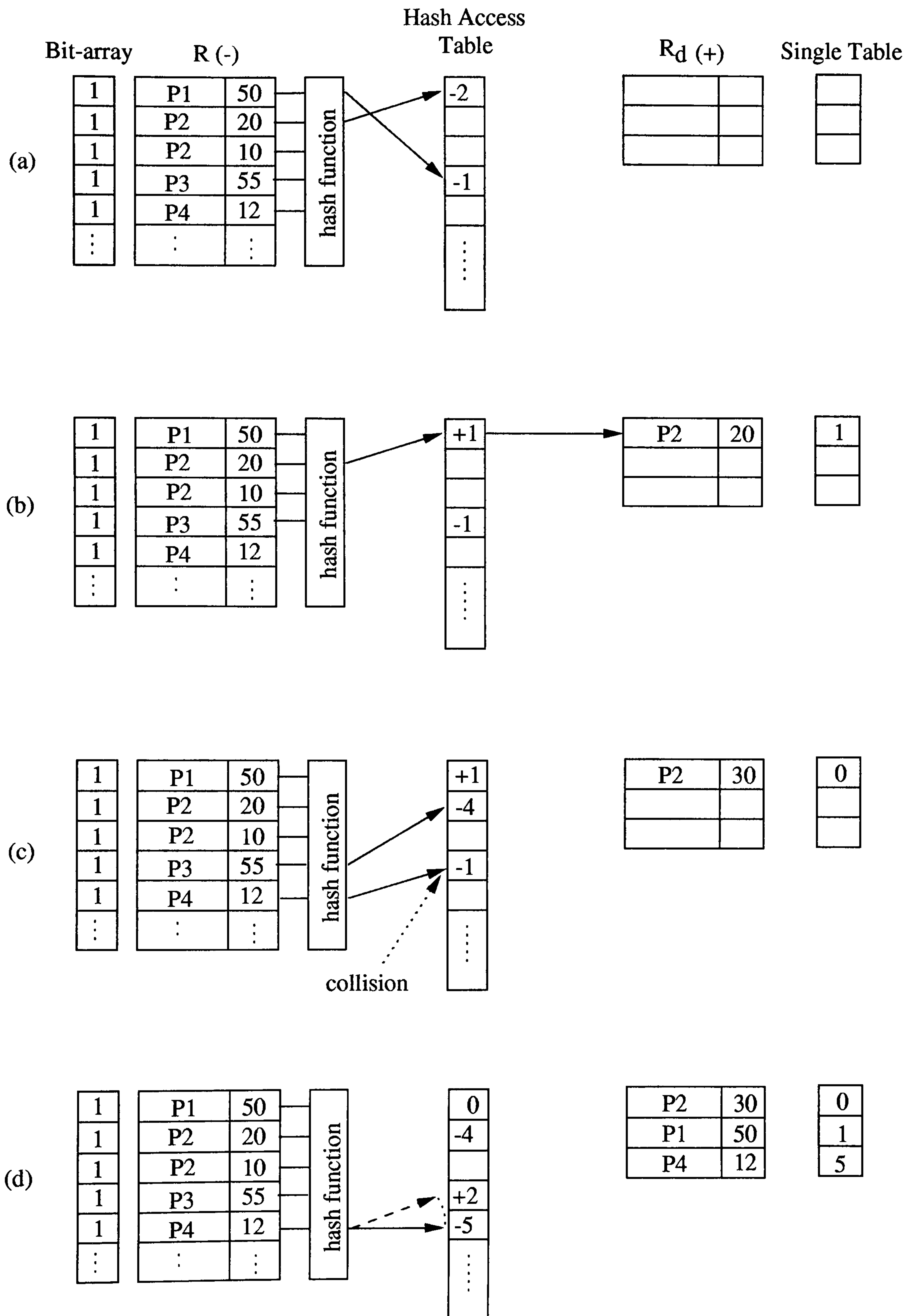

Figure 3.7: Extracting the differences during aggregation using the V-Aggregator

avoiding computation of aggregates already calculated. The Partially-Redundant views are thus processed and stored as compact materialised views. The abstract form of the proposed computation is shown in Figure 3.8, followed by the implementation of the L-R data cube.

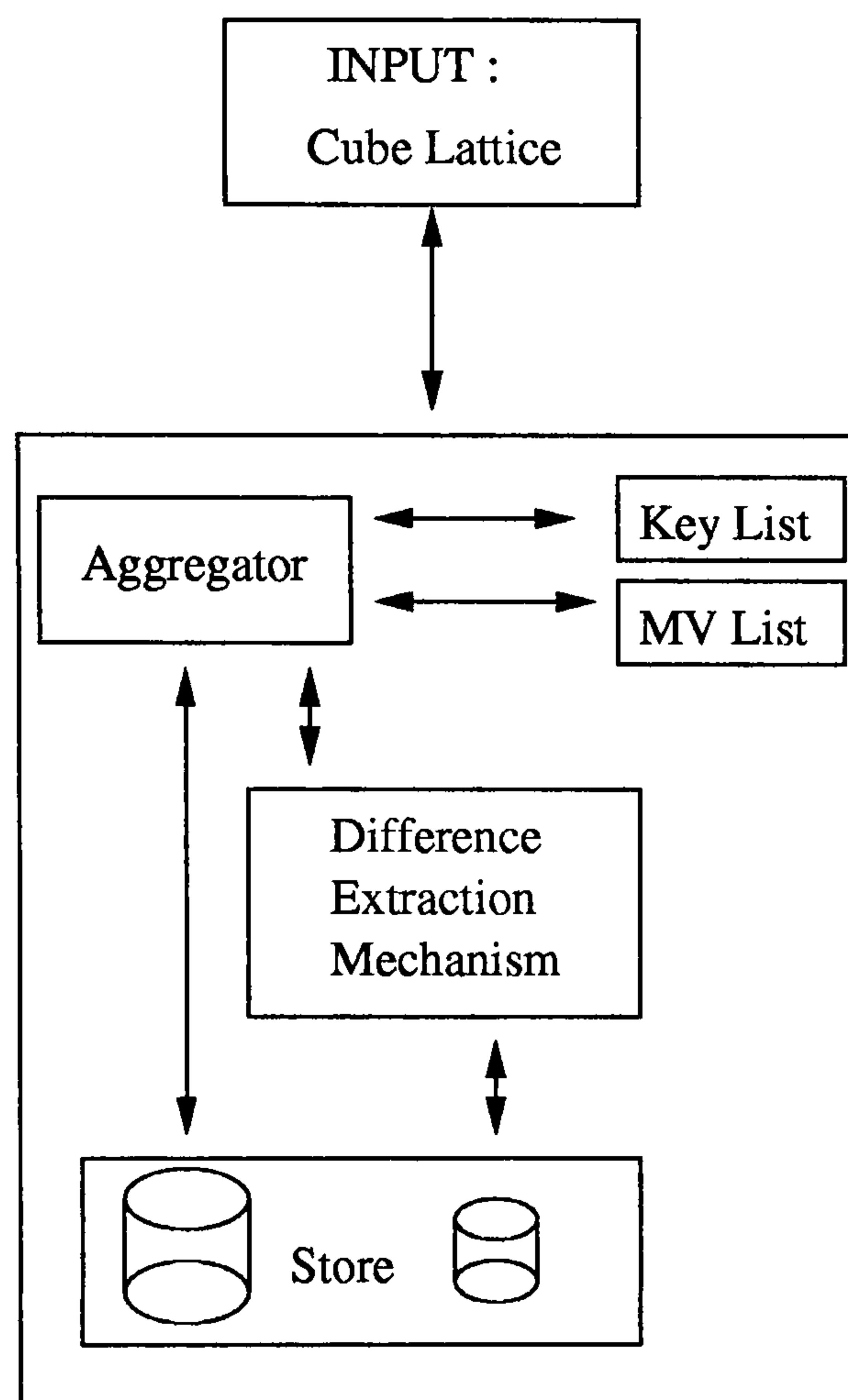

Figure 3.8: The L-R data cube

The Optimised Cube-By operator

Input: search lattice of the input Relation R

Set of Keys K, array of Keys

Output: Set of computed aggregates as views V

begin

$i := 0;$

$j := 0;$

$V := null;$

while $NoOfCombinations - 1 > i$ **do** (* From maximum to minimum arity *where* $NoOfCombinations = 2^n$ *)

begin

for j:=0 to entries in K

begin

if $K(j) \subset$ *GroupBy[i].schema* (* Totally redundant view *)

then

$i := i + 1;$

else (* not a Totally-Redundant view-candidate for computation *)

Aggregate(GroupBy[i]);

add *the GroupBy schema to* $V[i]$

$i := i + 1;$

end;

end;

return set V; (* the set of computed views *)

end;

Chapter 4

Experimental Confirmation

The objective of the experimental work is to verify the feasibility and scalability of the L-R methods and algorithms and prove that the new approach significantly improves the performance of OLAP systems with regard to space and time requirements. The experiments evaluated the new methods using a wide variety of real and synthetic datasets.

There are three groups of experiments. The first group relates to the performance of L-R in the computation of the full set of multidimensional aggregates (the data cube). The second relates to storage savings effected. The final group relates to query response times (or user's access time) from the already computed data cube. The three groups may be summarised as:

1. **Computing the Data Cube - performance timings**

2. **Storage of the Materialised Views - space savings**

3. **The Querying Response Time - performance timings**

4.1 The Experimental Configuration

The L-R algorithms were implemented as explained in Chapter 3. The tests were run on a Dual Pentium 200 MHz with 128 Mb of RAM and 1GB of virtual memory under Windows NT. There were no attempts to utilize the memory in a more efficient way than that provided by the operating system and also no attempt was made to utilize the second processor.

4.1.1 The Datasets

Both real and synthetic datasets were used in these experiments. There were four synthetic and two real datasets. Three of the synthetic datasets were taken from the TPC-D [TPC98] benchmark dataset and one from a hotel dataset [Kim96]. The real datasets were weather data [HWL94] and the adult dataset [Koh96]. Note that all datasets are considered sparse.

The TPC-D Datasets

The Transaction Processing Council (TPC) is an official benchmarking group supported by several hardware and database systems vendors. The TPC-D is a decision support dataset which can be generated at different scale factors defining the number of tuples in the fact table. In these tests, the lineitem table, from the TPC-D dataset, was used at three scale factors 0.1 (600K tuples), 0.01 (60K tuples) and 0.001 (6K tuples). In all datasets the measure of interest is the fifth attribute. The 6K dataset's attributes' cardinalities were: (1,500), (200), (10), (7), (50), (3), (2), (2,249), (2,234), (2,241), (4). For the 60K dataset, the attributes' cardinalities were: (15,000), (2,000), (100), (7), (50), (35,967), (2,520), (2,464), (2,531), (4), (7). The 600K dataset was generated with the scale factor sf=0.1 and the cardinalities were: (150,000), (20,000), (1,000), (7), (50), (2,526), (2,466), (2,548).

The Weather Dataset

This dataset is an archive of real weather data indicating cloud coverage over the ocean [HWL94]. The dataset contains (116,635) tuples with 20 dimensions. Ten of these were selected with the following cardinalities: (612), (2), (1,425), (3,599), (5), (1), (101), (9), (24), (10). The measure attribute was the sixth dimension.

The Adult Database

This is a real dataset from the US Census Bureau in 1994 which was first presented by [Koh96]. The original dataset size was 48,842 records with 14 attributes. Six of the attributes are numerical and the remaining six are categorical attributes. The down-loaded version contained 32,000 records and was projected to a relation with nine attributes, with the 9th attribute defined as the measured attribute. The cardinalities of the new dataset were: (72), (10), (19,988), (16), (16),(16), (7), (16), (7), (1). The last attribute was used for measure of interest.

The Hotel Dataset

This dataset, taken from a business example [Kim96], is a synthetic dataset with a fact table of Hotel Stays schema with eight dimensions and three measure attributes. The size of the dataset is 2,249 tuples. The small size was selected to ensure that the performance results in time were not effected by disk thrashing. The dataset and all its derivative aggregates, even in the conventional approach, almost fitted into the main memory. The cardinalities of the dataset were: (183), (26), (100), (20), (20), (2,168), (2), (20), (4), (907), (366) and (10).

4.2 Computing the Data Cube - Performance Timings

The computation of the data cube in L-R is performed in two stages (as explained in section 3.8).

- Stage 1: Determination of the set of Observational keys. In this stage, the Key-algorithm is responsible for the identification of non-redundant views, through the key extraction mechanism.
- Stage 2: Computation of the data cube by utilising the set of Observational keys (found in stage 1) to determine and eliminate Totally-Redundant views.

4.2.1 The Performance of the Key algorithm

Figure 4.1 compares timings for the key algorithm with the time required to compute the data cube using the conventional method [GBLP96] for the TPC-D 600K dataset. The experiment computed the data cube in different numbers of dimensions ranging from four to seven. The results show that on average the Key-algorithm only takes one tenth of the conventional time.

Figure 4.2 compares the timings between the key algorithm and the conventional time required to compute the data cube for the hotel dataset. The experimental configuration remains the same but now, in this test, it computes the data cube in different numbers of dimensions ranging from three to eleven.

The time required by the key algorithm is only of the order of 10% of that required to compute the data cube conventionally. Savings increase as the dimensionality increases.

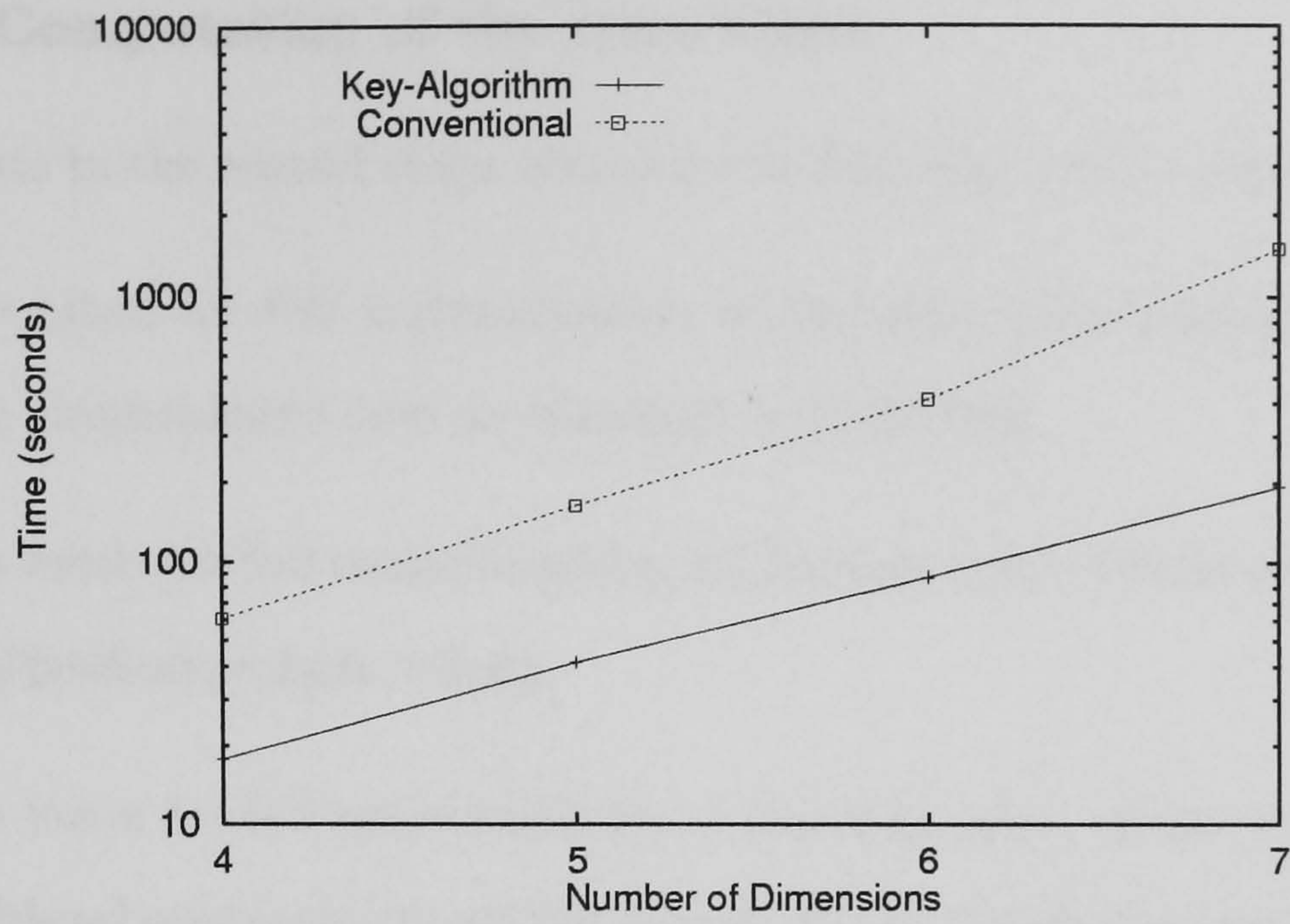

Figure 4.1: The conventional data cube time compared to the Key-algorithm time in the TPC-D 600K dataset

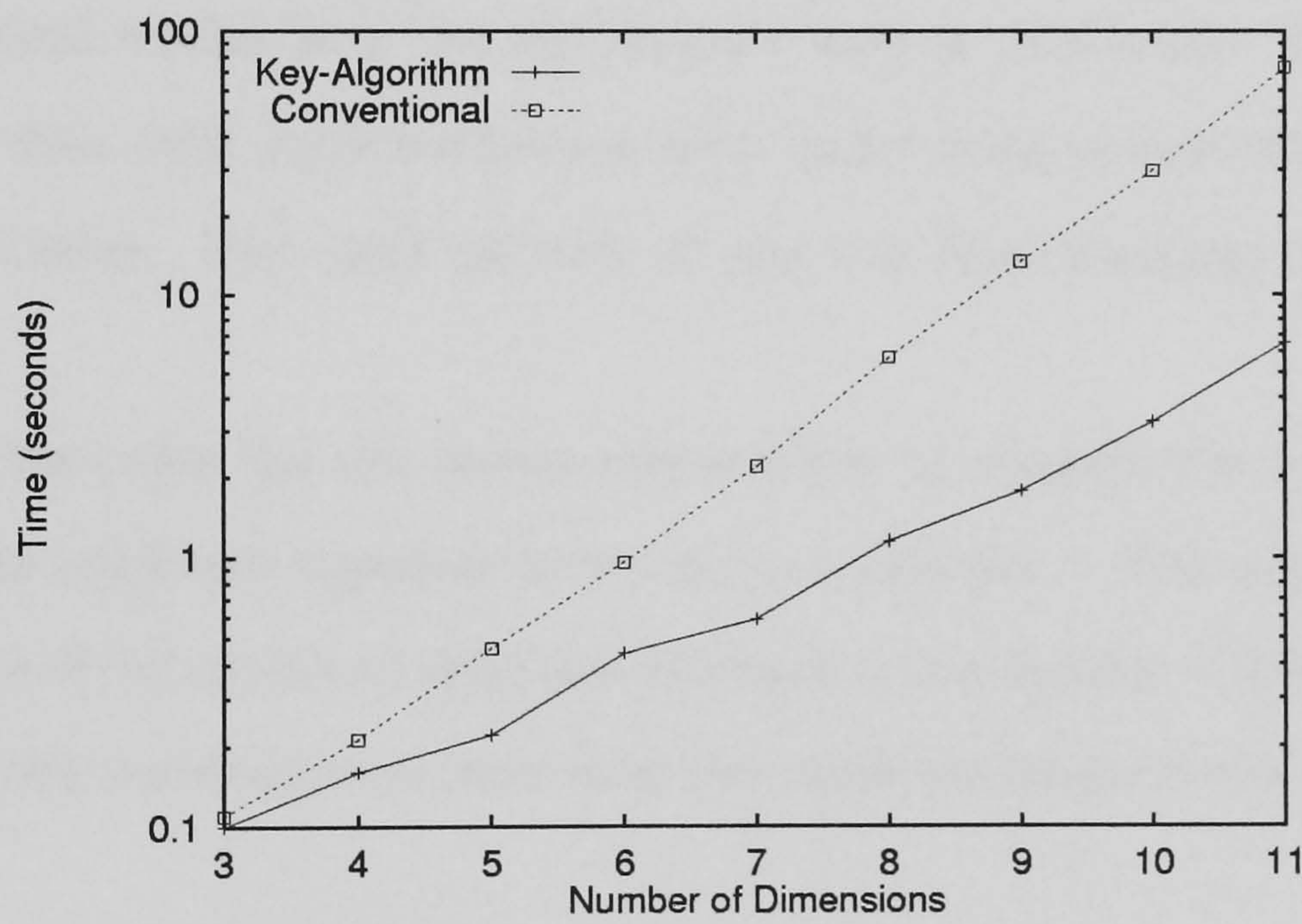

Figure 4.2: The conventional data cube time compared to the Key-algorithm time in the Hotel dataset

4.2.2 Full Computation of the Data Cube

The experiments in the second stage compare the following three timings:

- The time taken for full materialisation of the data cube using the **conventional** approach, implemented here as described in [GBLP96].
- The time taken for full materialisation of the data cube, which includes and utilises the **Totally-Redundant views.**
- The time taken for full materialisation of the data cube, which includes and utilises the **combined** approach (consisting of both Totally-Redundant and Partially-Redundant views).

Figure 4.3 and Figure 4.4 present averages taken from all datasets. Appendix C.1 presents analytical results from the six datasets used in this thesis. For each dataset, three different data cube implementations were tested using several dimensions varying from three to twelve. The total number of runs was approximately one hundred and seventy (170).

Figure 4.3 illustrates the two average times taken to compute the data cube conventionally over the combined approach in five and six datasets [1]. The results indicate that the performance of the combined approach increases as the number of dimensions increase and that after the ten-dimension data cube the combined outperforms the conventional computation.

[1]Because the tests for the 600K TPC-D dataset were run on up to seven dimensions, after the 7th dimension the average was taken from the remaining five datasets.

Figure 4.4 illustrates the ratio of the conventional to combined approach for computing a data cube of seven dimensions for all six datasets used in this thesis. The result shows that the combined approach in small dimensionality (seven dimensions) is slower than the conventional approach. The tests on the combined approach, however, utilised the B-aggregator algorithm whose performance is slower compared to the improved version (V-aggregator). The increase in savings with dimensionality indicates the scalability of the L-R approach.

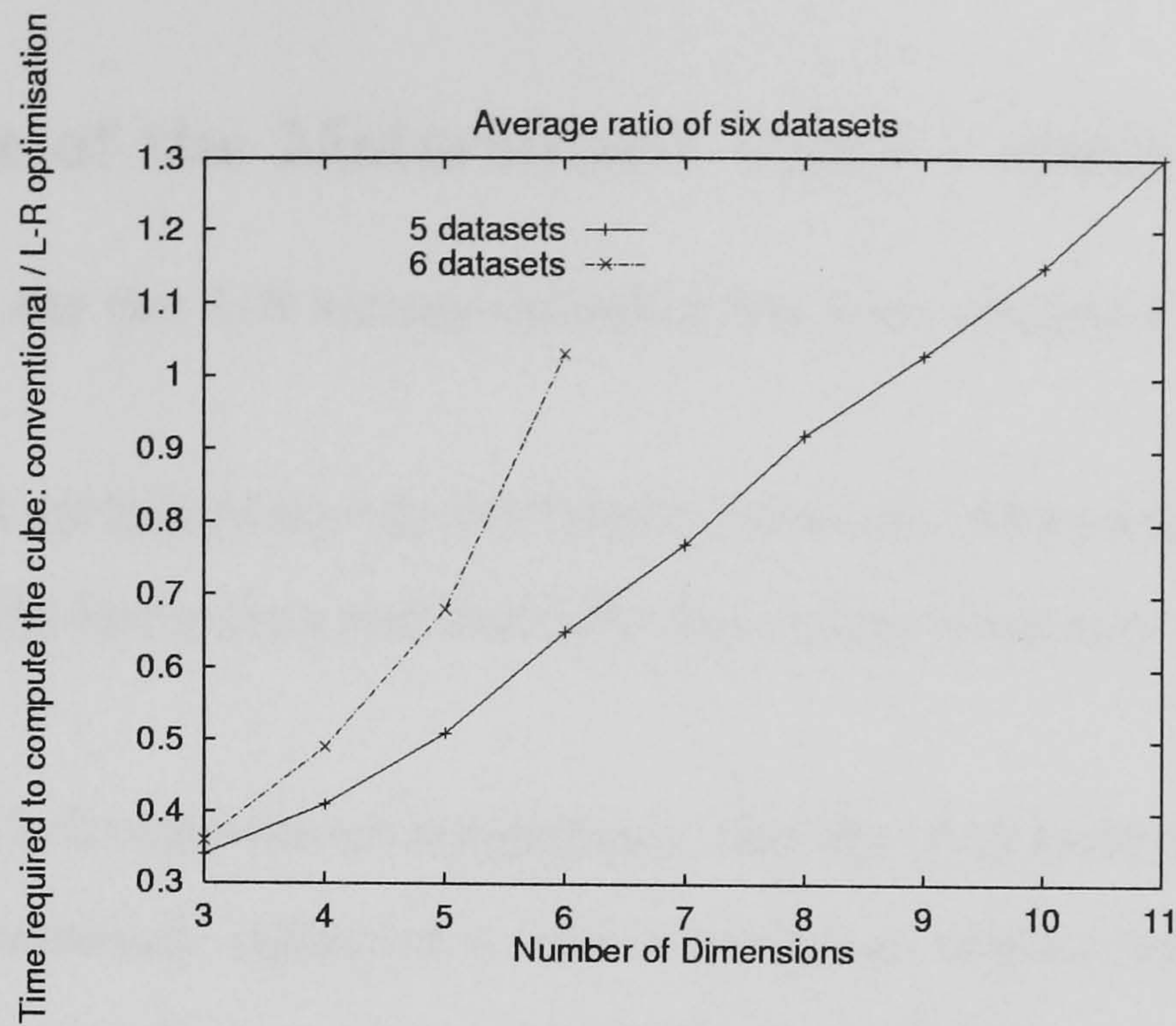

Figure 4.3: Time required to compute the data cube: conventionally over L-R, average of the six datasets

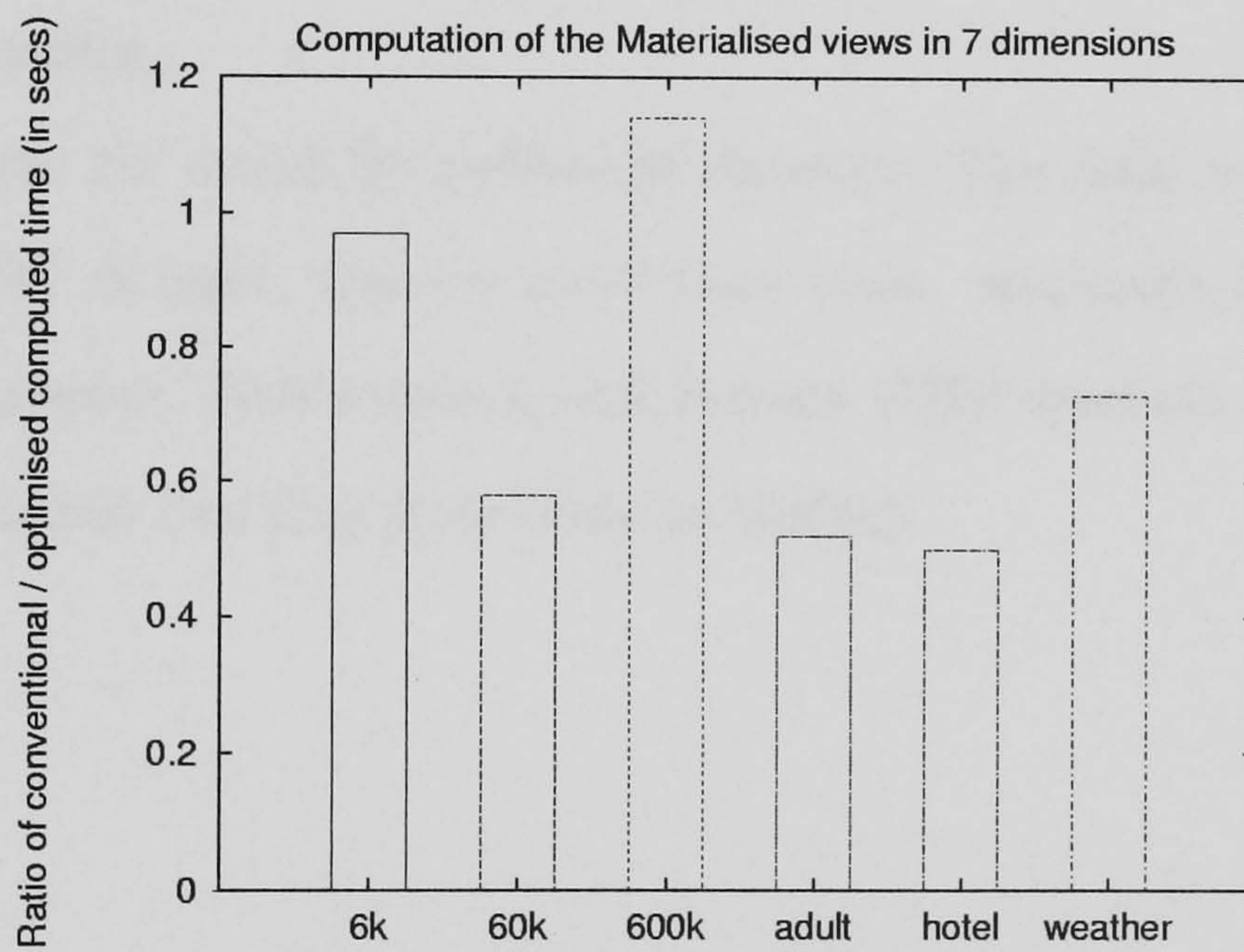

Figure 4.4: Performance in time of the L-R approach compared to the conventional approach in different datasets

4.3 Storage of the Materialized Views - space savings

For each dataset, the two L-R storage optimisations were compared to the conventional approach.

In the first L-R optimisation, only the Totally-Redundant views were utilised and in the second, both Totally-Redundant and Partially-Redundant views were utilised to eliminate redundancy.

The impact of L-R optimisation is significant. Savings of up to 30 times in space (adult dataset in 10 dimensions), compared to the conventional implementation, are achieved. Figure 4.5 and Figure 4.6 show average results for six datasets.

Figure 4.5 illustrates the average ratio of space required (conventional over L-R) in 6 datasets [2]. As expected, in all datasets, dimensionality is the crucial factor in the improving space savings.

Figure 4.6 shows the ratios for individual datasets. The ratio is never below six and at best, in the 60K dataset, rises to more than nine. Appendix C.2 shows analytical results from all datasets. One hundred and seventy (170) separate runs were conducted for different dimensions (varying from three to twelve).

[2]For the same reasons as explained earlier (section 4.2.2), the average time after the 7th dimension has been taken from five datasets.

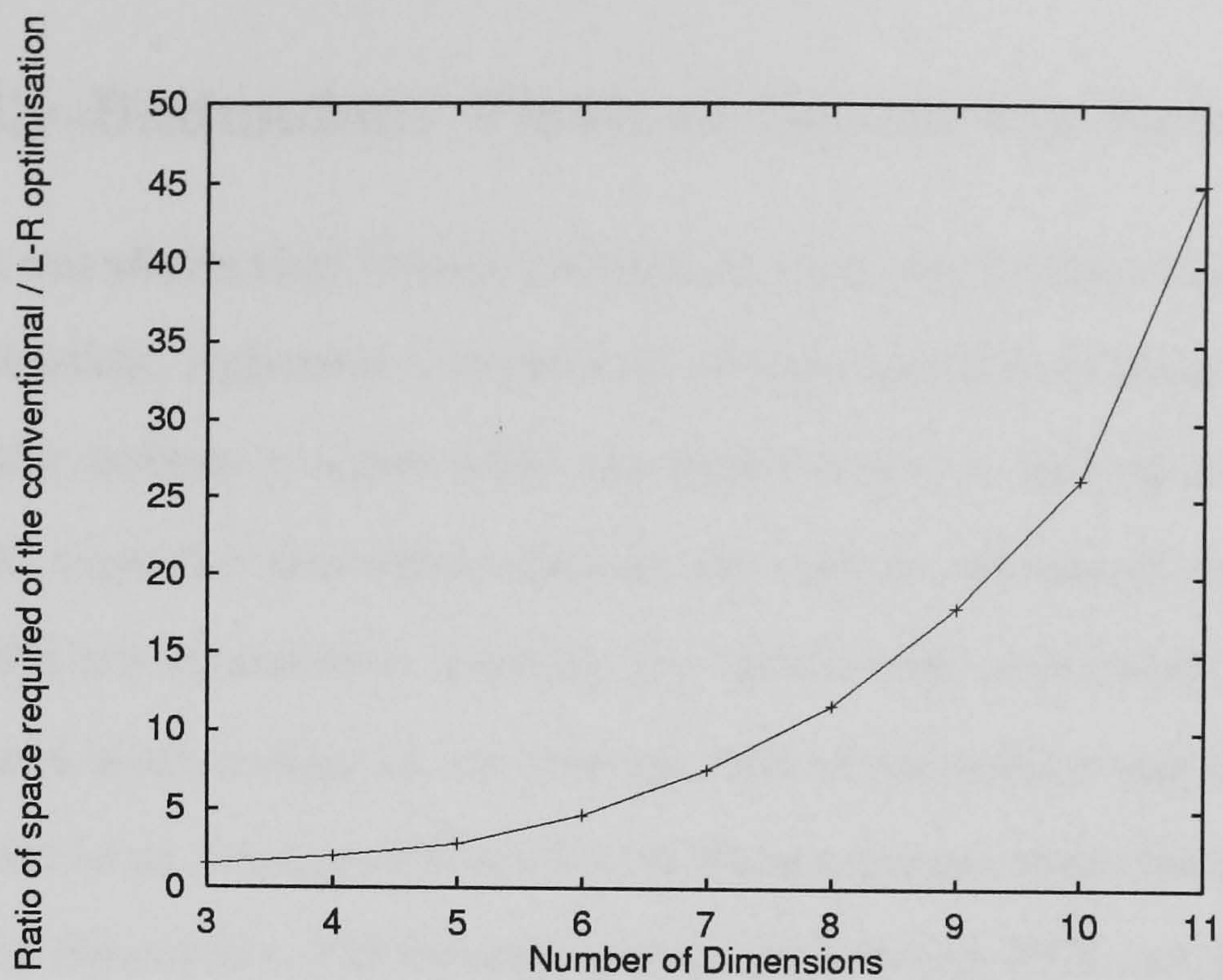

Figure 4.5: Space savings growth ratio in all datasets

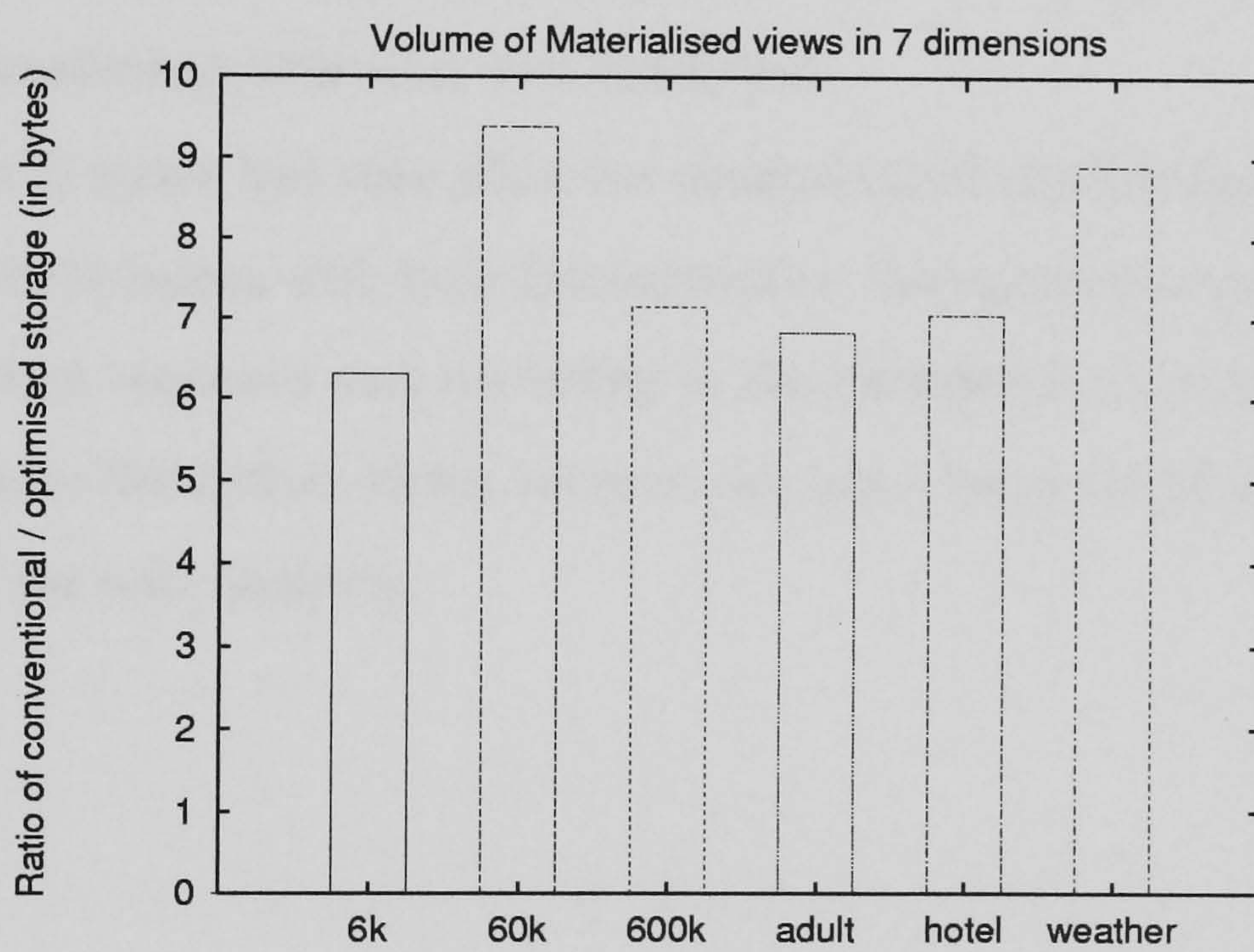

Figure 4.6: Performance in space savings of the L-R approach compared to the conventional approach in different datasets

4.4 Totally-Redundant Views of Derivative Relations

In Chapter 3, it was shown that Totally-Redundant views can be found in aggregates other than the base relation. Appendix C.3 presents all the experiments which were conducted and compares the savings in space when the base relation is used as a reference to the savings achieved when the derivative relations are used as references. On average, for a base relation with ten dimensions, applying the optimisation recursively to all derivative relations increased total savings to, on average, 90% of the total views in the data cube. This is in contrast to an average of 74% of total views achieved when the base relation was used. Using nine dimensions, the average savings are lower at 77% and 44% respectively. Appendix C.3 presents the detailed results from tests run in six datasets. For each dataset, the tests were run in several dimensions with the number varying from three to twelve. Altogether approximately fifty runs were conducted.

The savings in space and time after the elimination of the Totally-Redundant views are significant for relations with high dimensionality. Savings in storage and time for the Totally-Redundant approach vary according to the distribution and sparsity of the trial datasets. Partially-Redundant views, however, are much less sensitive to the distribution and sparsity of the trial datasets.

4.5 Query Response Time - Performance Timings

The query response time is the time taken by the database to respond to a user's query. The L-R methodology selects, computes and stores the materialised views in a compact form. The rationale of L-R storage optimisation is that it does not significantly slow the query response time compared to the conventional materialised view approach. This section will show that the L-R's retrieval time satisfies the above requirement, (i.e., fast retrieval of any aggregate). The experiments conducted for this purpose demonstrate that remarkable savings in storage space can be achieved without a significant trade-off in time. This group of experiments measured the time required to access aggregates stored in Difference form compared to the time required to retrieve the conventional materialised view. Both timings are compared to the 'on-the-fly' approach. Queries were made to aggregations varying from three to ten grouping attributes. The TPC-D 60K dataset was used with the following attributes: *Orderkey, Partkey, Suppkey, Linenumber, Returnflag, Linestatus, Shipdate, Commitdate, Receiptdate, Shipinstruct.*

Figure 4.7 compares five different measured timings for two different cases. Each timing was taken with eight different dimensionalities. In both cases, the result retrieved is a full aggregate relation and not a subset (resulting from a selective query). In the first case, the output group-by was returned as a 'deep-copy' of the aggregate (i.e., the deep-copy is a materialised view open to read/write operations).

In the second case, the resulting group-by is returned by reference or a 'shallow-copy' (i.e., the shallow copy is a view open to read-only operations). This eliminates additional storage and time to return the answer but it is slightly slower for subsequent accesses. Thus in the second case, the results are faster than the first. Note, in the second case the slightly slower performance of the combined approach compared to the materialised views (MV) approach occurs as a result of an operation which ensures that the system never

returns a 'view of a view' (refer to Section 3.4.1). The first three results relate to the first case in which the answer returns a deep copy of the group-by.

The timings shown in Figure 4.7 are taken from:

1. The on-the-fly computation. In this method it is assumed that only the base relation is available.

2. The combined approach. This is the time required to reconstruct the aggregation from its Difference representation. For this process, the parent relation and the Difference representation are stored in the backing store.

3. The conventional materialised view.

4. The combined approach. This is the time required to reconstruct the aggregation from its Difference representation. For this process the parent relation and the Difference representation are present in the main memory.

5. The conventional materialised approach when all the materialised views are present in the main memory.

The results indicate the benefits of pre-computation of the multidimensional aggregates; the materialised views are faster but the data volume required is much higher (27 times larger for the TPC-D lineitem dataset in 10 dimensions) than the combined L-R approach. The 'on-the-fly' approach is always slower, as was expected. Both the combined and the materialised views in the 'by-reference mode' are faster than their retrieval from secondary storage. The advantage of the L-R approach is that its compact storage enables the data to be retained in the main store. This implies that the majority of the aggregates will be restored from their stored-in-memory parent and thus the speed will be equal to that of the fourth timing (see Figure 4.7). The large volume of conventional materialised views prohibits in-memory aggregation and thus the majority of the aggregates will operate through secondary storage.

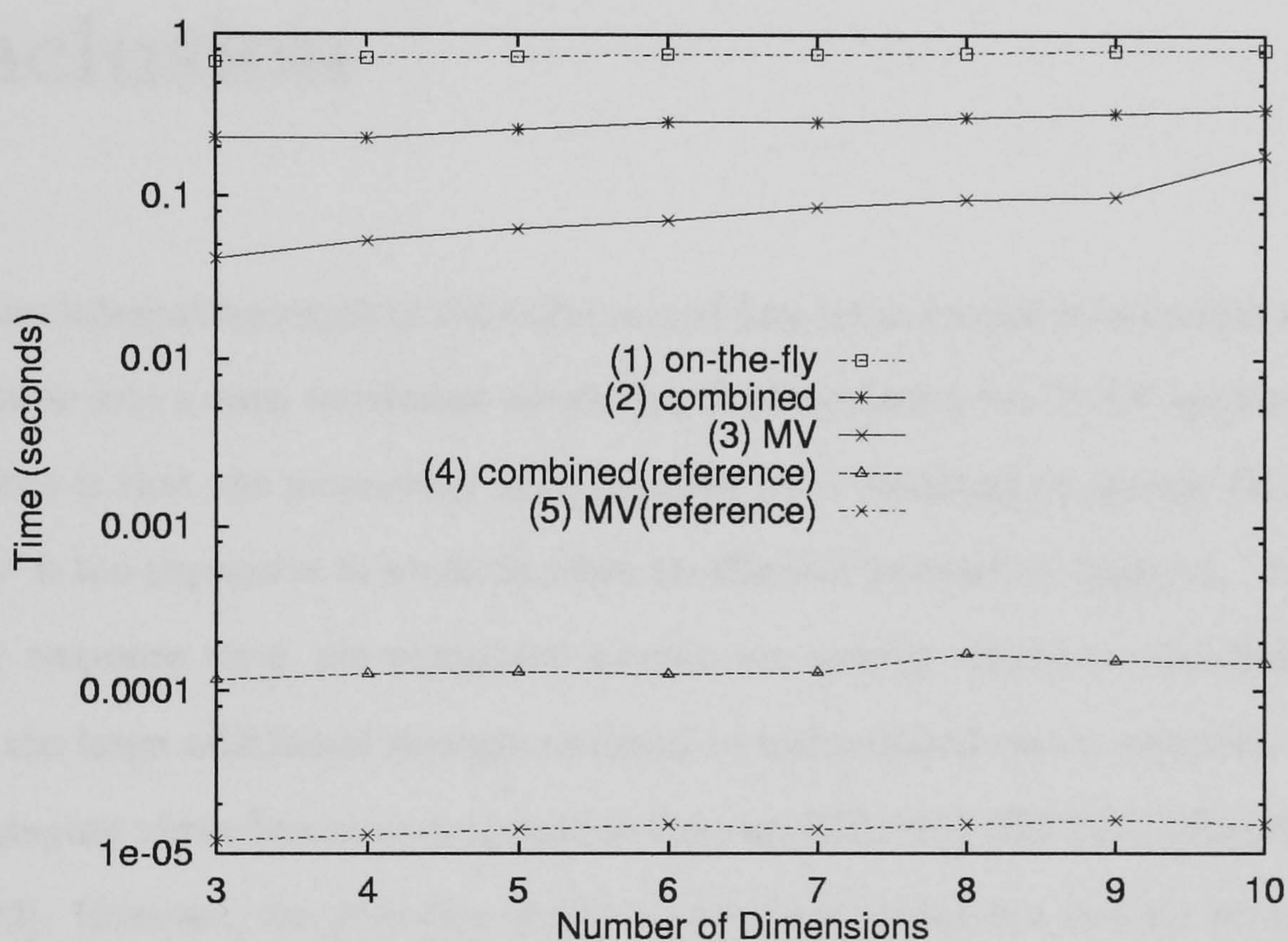

Figure 4.7: Group-by's response time

Chapter 5

Conclusion

Information integration requires the collection of data from several information sources and incorporation into a data warehouse which lays the foundation for OLAP applications. The key problem is that the processing time required for a database to answer OLAP queries 'on the fly' is too expensive to accommodate an effective interactive dialogue. To accelerate the query response time, pre-computed queries are usually stored as materialised views. To avoid the large additional storage overhead of materialised views, selection of a subset of the aggregate views has been proposed in the past [HRU96], [BPT97], [Gup97], [SDN98] and [BR99]. However, the selection of the appropriate subset is a crucial issue.

This thesis contributes to the fundamental understanding of the nature of the data cube process and has introduced the Low-Redundancy (L-R) approach. The proposed L-R approach is novel and differs from any previous approach [HRU96], [BPT97], [Gup97], [SDN98] and [BR99]. The approach achieves fast computation and compact storage of the aggregates through methods based on extending traditional relational theory to the OLAP environment. Specifically, this work has shown that:

- Many of the possible aggregates are directly derivable from the parent input (base) relation without any processing. These aggregates are called Totally-Redundant views and a new formalism, derived from relational theory, provides a means of determining which views belong to this category only by inspection, avoiding additional processing cost. The practical implication of this is that a large percentage of views require no processing or storage (e.g. 44% - on average for all datasets used in this thesis in 9 dimensions - directly from the input (base) relation). Further savings of up to 77% are achieved - on average for all datasets in 9 dimensions - when the optimisation is applied to the whole set of the derivative aggregates (not merely to the input (base) relation).

- Aggregates not belonging to the set of Totally-Redundant views are classified as Partially-Redundant. A subset of each Partially-Redundant view is g-equivalent to a subset of, and thus derivable from, its parent relation. Hence only those tuples which are different from those in its parent relation need be stored resulting in remarkable savings in space. Typically, the Partially-Redundant views require 30 times less space than those stored conventionally.

- The L-R approach has been evaluated with the set of algorithms provided in this thesis and the experimental work has demonstrated that the approach provides an efficient, practical and scalable methodology for OLAP systems.

- The new approach is independent of the structure of data and can be applied to either ROLAP and MOLAP systems and can be also integrated with other existing techniques such those described in Chapter 2.

- At the querying stage, the retrieval of any aggregate must appear almost as fast as a materialised views. The reconstruction of any aggregate from its Difference representation satisfies this criterion, with only a small cost in memory for each view.

5.1 Implications of the L-R approach

The implications of the L-R approach will be discussed in the following sections. The L-R approach positively affects indexing, the user interface and the main store.

5.1.1 Indexing in OLAP

Randomly accessing records in large relations typically requires indices. [Ram98] defines the index as an auxiliary data structure designed to speed up operations which are not efficiently supported by the basic organisation of records in that file. Consider the relation R which is a set of products $P1, P2, .., Pn$ in a company and the following query:

SELECT *
FROM R
WHERE $Product = P2$

The above query requires scanning of the whole relation to retrieve the tuples on product $P2$. Having an index for the relation R would speed-up the searching of qualifying tuples.

In OLAP databases, indexing is an important issue since the materialised views have to be indexed to accelerate access to them. The volume of materialised views in an OLAP database and the additional overhead from indices, results in very expensive systems in storage terms. [Rous82] and [GHRU96] have analysed the topic of the index for

materialised views. The expansion of multidimensional aggregates follows 2^n combinations in which the number of subsets of a set with n-elements is 2^n. There can be several indices in a view and the number of indices varies according to the number of attributes in the view. The effect of the L-R approach on indexing is significant. Totally-Redundant views avoid any indexing overhead since they are never stored. Accessing tuples from a Totally-Redundant view is achieved by pointing to the index of a parent stored relation. Thus Totally-Redundant views can utilise the index of their parent views.

Partially-Redundant views, as described earlier in section 3.2, contain two subset views, one with the g-equivalent tuples and a second with the aggregate tuples. The view with the g-equivalent tuples requires no indexing. For the second view - the one with the Differences - the construction of an index is necessary. However, the index size in this view is smaller than the whole Partially-Redundant view.

5.1.2 The User Interface

The potential of the L-R approach is significant in the user interface. For OLAP navigation, it is important to reduce the time spent by the user extracting useful information from the data. Browsing views can be very effective when it supplies the user with information regarding redundant views. For example, if a view is 'Totally-Redundant' then it needs no exploration. By supplying the user with this information, it facilitates faster navigation and exploration through the aggregates. The effect is more obvious in cases with large dimensionality datasets. For example, a sixteen dimension dataset would require 65,356 views to explore it. For obvious reasons, the user is unable to navigate through all of these views. The L-R enhancement thus provides the means for faster exploration of aggregate views since approximately 50% of the views (TPC-D 60 K) are Totally-Redundant.

The implementation of a user interface could be enhanced by integrating tools to

improve the navigation process. Such tools could be based on: audio, visual, colour and graphs with fewer vertices and edges than the original (see Figure 5.1). Further exploitation of the 'Partially-Redundant' views could reveal useful information about the similarity between groups of tuples. Figure 5.1(a) shows the graph of a three dimensional cube and its transformation to the low-redundant cube in Figure 5.1(b). The solid lines denote the edges of the graph which represent the unique aggregates. The dashed lines denote the paths which have been eliminated due to g-equivalent views (in circle).

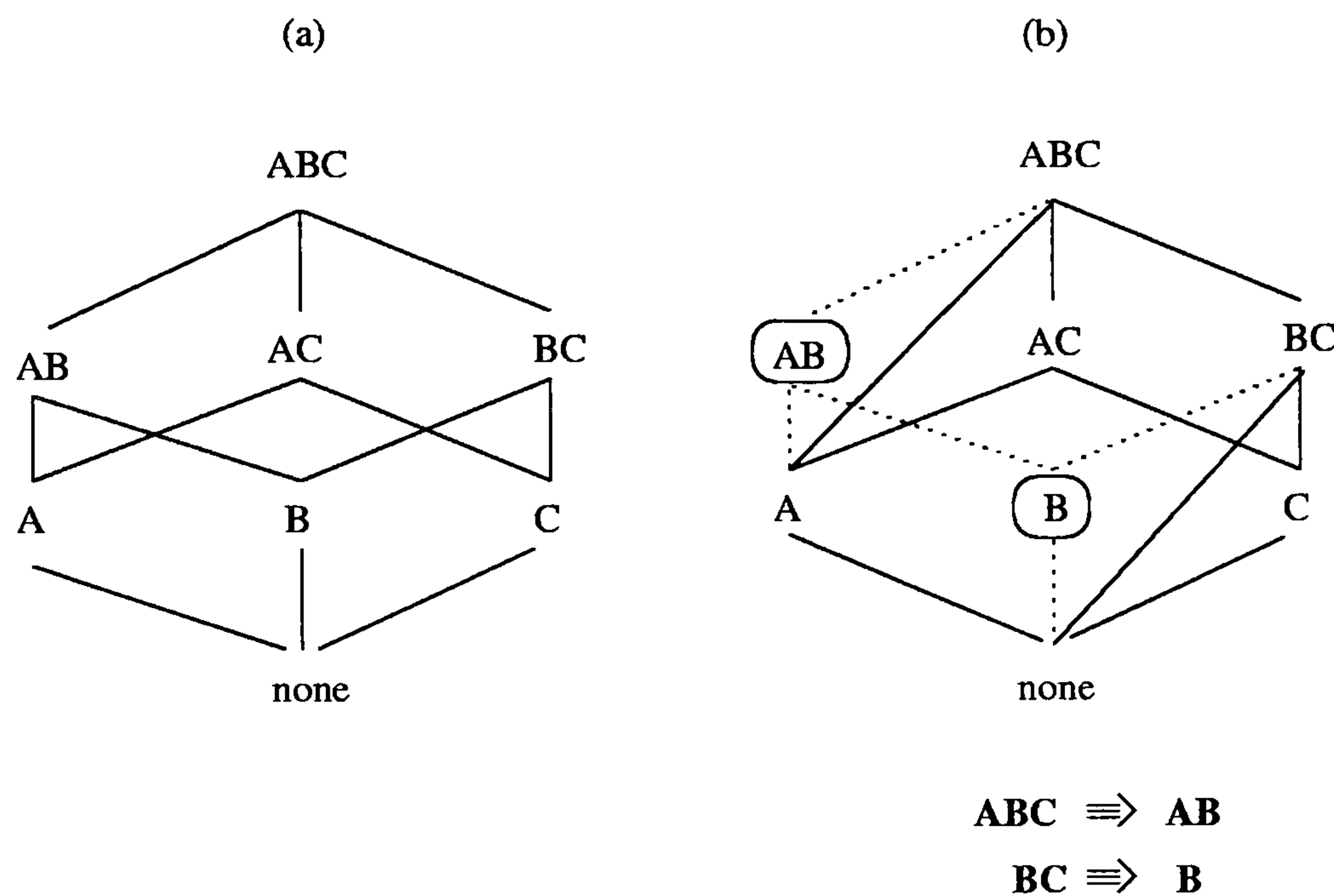

Figure 5.1: Elimination of Totally-Redundant views

5.1.3 The Main Store

Data warehouses typically are large repositories of data which are stored in order of Petabytes (10^6 *Gigabytes*). [FSM91] observed that data doubles in size every twenty months. Conventional implementations of the data cube incorporate algorithms to partition data into the secondary store and transfer them to main memory for computation [RS97], [BR99]. In these implementations, the goal is to utilise the main store to the maximum.

Those systems which adopt secondary storage techniques can take advantage of the L-R method to reduce I/O traffic, allowing more data to be retained in the cache or main memory. The important feature of the L-R approach is that its performance in time and space savings increases as the number of dimensions increase. Thus, for high dimensionality datasets, L-R will outperform conventional implementations. Hence, the benefit of L-R is in retaining faster RAM-accelerated performance while also reducing the required RAM storage.

The methods proposed in this thesis are based on a conceptual data model and not on special data structures. In Chapter 3 it was indicated that the approach is based on the redundancy of aggregate views or tuples in the database, so any fast data processing technique could be combined with the L-R approach. The compatibility of L-R with other approaches for selection and computation of multidimensional aggregates is a further strong advantage.

5.2 Future Work

The future work of this research could be to explore the potential of the new L-R approach in areas other than the implementation of the data cube. These are:

- OLAP and data mining which are closely related research areas and support the same group of users. The desire to extract useful information from the data has introduced new methodologies to the knowledge discovery process. [FSM91] defines knowledge discovery in databases as the non-trivial process of identifying valid, potentially useful and ultimately understandable patterns of data. Data mining, as part of the knowledge discovery process, searches for patterns of interest in a particular representational form or a set of such representations [FSSU95]. The L-R methodology identifies and extracts the non-redundant groups of views or groups of tuples as was described in Chapter 4. The importance of redundancy in multidimensional

aggregates from an information perspective is an area which could be investigated. Since the L-R approach is the only method of identifying redundancy in the multidimensional aggregates, it could provide meaningful information which may be of significant benefit to the user.

- The implications for the user interface need further research. The evaluation of a 'new interface' compared to existing implementations would reveal the importance of the new method.

- The compatibility of the L-R with other techniques should also be investigated. This area of work would focus on finding possible integrations with methods for selection, computation of the data cube and storage of the aggregates as materialised views.

Bibliography

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. *In Proceedings of the 22nd International Conference on Very Large Databases*, pages 506-521, Mumbai, Sept. 1996.

[Babb79] E. Babb. Implementing a Relational Database by Means of Spesialized Hardware. *In ACM Transactions on Database Systems, Vol.4, No. 1*, March 1979, Pages 1-29.

[BZ98] C. Bontempo, G. Zagelow. The IBM Data Warehouse Architecture. *In Communication of the ACM* 41(9):38-48, 1998.

[Bloom70] B.H., Bloom. Space/Time trade-offs in hash coding with allowable errors. *In Communication of the ACM* 13(7):422-426.

[BPT97] E. Baralis, S. Paraboschi, E. Teniente. Materialized View Selection in a Multidimensional Database. *In Proceedings of the 23rd International Conference on Very Large Databases*, pages 156-165, Athens 1997.

[BS98] D. Barbara, M. Sullivan. Quasi-Cubes: A space-efficient way to support approximate multidimensional databases. Technical Report, Department of Information and Software Engineering, George Mason University 1998.

[BR99] K. Beyer, R. Ramakrishnan. Bottom-Up Computation and Iceberg CUBEs. *In Proceedings of the ACM SIGMOD International Conf. on Management of Data*, pages 359-370, Philadelphia PA, USA, June 1999.

[Codd70] E.F. Codd. A relational model for large shared data banks. *Comm. ACM, 13(6):377-387, 1970.*

[Codd93] E.F. Codd, S.B. Codd, C.T. Salley. Providing OLAP (On-Line Analytical Processing) to User Analyst: An IT Mandate. *Arbor Software* at http://www.arborsoft.com/OLAP.html.

[CS94] S. Chaudhuri, K. Shim. Optimizing queries with aggregate views. *In Proceedings of the Extending Database Technology (EDBT)* pages 167-182, 1996.

[CD97] S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP Technology. Technical Report MSR-TR-97-14, Microsoft Research Advanced Technology, Redmond March 1997.

[CKL+97] L. Colby, A. Kawaguchi, D. Lieuwen, I.S. Mumick, K. Ross. Supporting multiple view maintenance policies: concepts, algorithms and performance analysis. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[DANR96] P.M. Despande, A. Shukla, J.F. Naughton, K.Ramaswamy. Storage Estimation of the Multidimensional Aggregates. *In Proceedings of the 22nd International Conference on Very Large Databases*, pages 522-531, Mumbai, Sept. 1996.

[FSM91] W.J. Frawley, G. Piatetsky-Shapiro, C.J. Matheus. Knowledge Discovery in Databases: An Overview. In *Knowledge Discovery in Databases*, eds. G. Piatetsky-Shapiro and W.J. Frawley. pages 1-27. Menlo-Park, California: The AAAI press.

[FSSU95] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusami. Advances in Knowledge Discovery and Data Mining. AAAI Press / MIT Press.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases, *ACM Computing Surveys* 25(2):73-170, June 1993.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Technical Report MSR-TR- 95-22, Microsoft Research Redmond 1995.

[GHRU96] H. Gupta, V. Harinarayan, A. Rajaraman, J.D. Ullman. Index Selection for OLAP. In *Proc. of the 13th ICDE*, pages 208-219, Manchester, UK, 1997.

[Gupt97] H.Gupta. Selection of Views to Materialize in a Data Warehouse. *In Proceedings of the 6th International Conference in Database Theory (ICDT)*, pages 98-112, Delphi, Jan 1997.

[GHQ95] A. Gupta, V. Harinarayan, D. Quass. Aggregate-query processing in data warehousing environments. *In Proceedings of the* 21st *International Conference on Very Large Databases*,pages 358-369, Zurich, Switzerland 11-15 1995.

[GM99] A. Gupta, I.S. Mumick. Materialised Views - Techniques, Implementations and Applications. *The MIT Press 1999.*

[Hans99] E.N. Hanson. A performance analysis of view materialization strategies. In Materialised Views - Techniques, Implementations and Applications. pages 511-533 *The MIT Press 1999*

[HRU96] V. Harinarayan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently. *In Proc. ACM SIGMOD International Conference on Management of Data*, pages 205-227, 1996.

[HWL94] C.J. Hahn, S.G. Warren, J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. *Available from http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html.*

[Joh97] J. L. Johnson. Database-Models, Languages, Design. pages 90-91, Oxford University Press Inc. 1997.

[Kim96] R. Kimball. The Data Warehouse Toolkit. John Wiley, 1996.

[Klug82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *In Journal of the ACM, 29(3):699717*, 1982.

[Knu98] D.E. Knuth. The Art of Computer Programming. Volume 3, Second edition, Addison Wesley 1998.

[KM99] N. Kotsis, D.R. McGregor. Compact Representation: An Efficient Implementation for the Data Warehouse Architecture. *In Proceedings of the 1st International Conference in Data Warehousing and Knowledge Discovery (DAWAK)*,pages 78-85. Florence, Italy, August 1999.

[Koh96] R. Kohavi. Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining,1996.

[KR82] G.P. McKeown, V.J Rayward-Smith. Mathematics for Computing. *The McMillan Press ltd.* Pages 349-355,1982.

[LQA97] W. J. Labio, D. Quass, B. Adelberg. Physical Database Design for Data Warehouses. *In Proceeding of the International Conference on data Engineering (ICDE)*, 1997.

[LO78] C.L. Lucchesi, Sylvia L. Osborn. Candidate Keys for Relations. Journal of Computer and System Science 17, pages 270-279 (1978).

[ME92] P. Mishra, M.H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys* 24(1):63-113, March 1992.

[ML86] L.F. Mackert, G.M. Lohman. R* Optimizer: Validation and performance evaluation for distributed queries. *In Proceedings of Conference on Very Large Databases* pages 149-159, 1986.

[MTD76] D.R. McGregor, R.G. Thomson, W.N. Dawson. High performance hardware for database systems. Appeared on *Systems for Large Data Bases.* Editors P.C. Lockemann and E.J. Neuhold. North-Holland Publidhinh Company, 1976.

[MUW99] H. Garcia-Molina, J.D. Ullman, J. Windom. Database System Implementation. Prentice Hall, Upper Saddle River, New Jersey 07458, 1999.

[MQM97] I.S.Mumick, D. Quass, B.S. Mumick. Maintenance of summary tables in warehouse. *In Proceedings of ACM SIGMOD 1997 International Conference on Management of Data*, Tucson, AZ, May 1997.

[OOM87] G. Ozsoyoglu, M. Ozsoyoglou, V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *In ACM transactions on Database Systems*, 12(4):566-592, 1987.

[OG95] P. O'Neil, G. Graefe. Multi-Table Joins through Bitmapped Join Indices. *In Proc. ACM SIGMOD International Conference on Management of Data, 1996.*

[O87] P. O'Neil. Model 204 Architecture and Performance. *Springer-Verlag LNCS359, 2nd Intl. Workshop on High Performance Transctions Systems* Asilomar, CA, Sept 1987.

[Pen99] N.Pendse. Database explosion available at *www.olapreport.com/ Database Explosion.htm.*

[Ram98] R. Ramakrishnan. Database Management Systems. *WCB/McGraw Hill*, 1998.

[RS97] K.A. Ross, D.Srivastava. Fast Computation of Sparse Datacubes. *In Proc. of the 23rd International Conference on Very Large Databases*, pages 116-125, Athens 1997.

[RSC97] K. Ross, D. Srivastava, D. Chatziantoniou. Quering Multiple Features of Groups in Relational Databases. *In Proceedings of the 22nd International Conference on Very Large Databases*, pages 295-306, Mumbai, Sept. 1996.

[Rous82] N. Roussopoulos. View Indexing in Relational Databases. *In ACM Transactions on Database Systems.* Vol. 7, NO. 2, June 1982, Pages 258-290.

[Rous97] N. Roussopoulos. Materialised Views and Data Warehouses. Technical Report, Department of Computer Science and Institute of Advanced Computer Studies, University of Maryland, 1997.

[SAG96] S. Sarawagi, R. Agrawal, A. Gupta. On Computing the Data Cube. *Research report 10026, IBM Almaden Research Center*, San Jose, California, 1996.

[SDN98] A. Shukla, P.M. Despande, J.F. Naughton. Materialized View Selection for Multidimensional Datasets, *In Proceedings of the 24th International Conference on Very Large Databases*, pages 488-499, New York 1998.

[SDJL96] D. Srivastava, S. Dar, H.V. Jagadish, A.Y. Levy. Answering Queries with Aggregation Using Views. *In Proceedings of the 22nd Conference on Very Large Data Base* pages 318-329. Mumbai(Bombay), India 1996.

[Sel88] T. Sellis. Multiple query optimization. *In ACM Transactions on Database Systems*, 13(1):23-52, 1998.

[Schn97] D. Schneider. The Ins & Outs (and everything in between) of Data Warehousing. Tutorial in 23rd International Conference on Very Large Data bases. Athens, Greece 1997.

[SL76] D.G. Severance, G. Lohman. Differential filesL Their application to the maintenance of large databases. *ACM Transactions on Databases Systems (TODS)*, 1(3):256-267, September 1976.

[TPC98] F. Raab, editor. TPC Benchmark TM D Standard Specification Revision 1.3.1 Transaction Processing Council 1998

[VW99] J.S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 193-204, Philadelphia PA, USA, June 1999.

[Wid95] J. Widom. Research Problems in Data Warehousing. *In Proceedings of the 4th International Conference of CIKM*, pages 25-30, Nov.1995.

[WB98] M.C. Wu and A. Buchmann. Encoded bitmap indexing for data Warehouses. *In Proceedings of the International Conference on data Engineering (ICDE)*, pages 220-230, 1998.

[YL95] W. Yan, P. Larson. Eager aggregation and lazy aggregation. In *Proc. of the Twenty-First International Conference on Very Large Databases (VLDB)*, pages 345-357, 1995.

[ZDN97] Y. Zhao, P.M. Deshpande, J.F. Naughton. An Array based Algorithm for Simultaneous Multidimensional Aggregates, *In Proc. of the ACM SIGMOD International Conference On Management of Data*, pages 159-170, 1997.

APPENDICES

Appendix A

Source code: L-R aggregation

```
{Author: Nikolaos Kotsis
 Synopsis: The main algorithm to perform an aggregation and
             also extracts the Difference representation  }

function TRelation.Aggregate(aFirstGroupBy:Boolean;aCountBool:Boolean;
aKeyPresent:boolean):TRelation;
var
i,j,k,m,n,tS,step,hNo,c,y,z:integer;
found:boolean;
tupleI,tupleJ:TTuple;
vAccessTable: array of integer;
vSingleTable,vHashNo: array of integer;
relDiff:TRelationDiff;
x:TRelation;
newRelation:TRelationStored;
```

```
bset3,bset4:TBitset;
bset1,bset2:ThashSet;
begin
    if not aKeyPresent then
    begin
        n:=self.NoOfTuples;
        bset1:=Thashset.Create;
        bset2:=Thashset.Create;
        bset3:=TBitset.Create;
        bset4:=TBitset.Create;
        newRelation:=TRelationStored.init(self.tupleSize,self.noOfTuples,
        self.schema,self.dict);
        newRelation.measureCol:=self.schema.indexOf(measureName)+1;
        newRelation.measureName:=measureName;
        {self is view of oldRelation}
        ts:=tupleSize;
        setLength(vAccesstable,(3*n)+1);
        setLength(vHashNo,n+1);
        m:=0;
        TupleI:=self.get(1); TupleJ:=self.get(1);
       {Find B2 the set of tuple hash values which are held
      by more than one tuple}
        for i:=1 to n do
        begin
            self.getTuple(i,TupleI);
            hNo:=tupleI.hashDims;  vHashNo[i]:=hNo;
```

```
          if not bset1.inset(hNo) then
          {First time this hash value has appeared}
          bset1.insert(hNo)
          {At least one previous tuple has this hash value}
          else bset2.insert(hNo);
     end;
     m:=0;
     setLength(vSingleTable,n+1);
     for i:= 1 to n do
     begin
          hNo:=vHashNo[i];
          if bset2.inset(hNo) then
          begin
               self.getTuple(i,TupleI);
               found:=false;
               j:=(hNo mod (3*n));
               step:= hNo mod 13 + 1;
               k:=vAccessTable[j];  c:=0;
               while (not found)and (k<>0)  do
               begin
                    newRelation.getTuple(k,tupleJ);
                    c:=c+1;
                    {Check is the ith tuple found in the m tuples
                    already being used by the aggregation}
                    if tupleI.equalFields(tupleJ,ts-1) then
                    begin
```

```
                {Existing aggregation found}
                found:=true;
                if countIs1 {or (aCountBool and aFirstGroupBy)}
                then y:= 1
                else
                y:=strToInt(dictionary.stringForToken(TupleI.get(ts)))
                newRelation.A.put(k,ts,(newRelation.A.get(k,ts) + y));
              vSingleTable[k]:=0;
            end
            else
            begin
             if c<=5 then j:=(j+step) mod (3*n)
                else
                j:=(j+1) mod (3*n);
      k:=vAccessTable[j];
            end;
         end;
         if not found then
         begin
             m:=m+1;
             vAccessTable[j]:=m;
             vSingleTable[m]:=i;
             {mapping from the view to the parent relation}
             for k:=1 to tS-1 do
             newRelation.A.put(m,k, tupleI.A[k]);
             if countIs1 {or (aCountBool and aFirstGroupBy)} then
```

```
                begin
                  newRelation.A.put(m,ts{newRelation.TupleSize},1)
                end
                else
                begin
                    newRelation.A.put(m,ts{newRelation.TupleSize},
                    strToInt(dictionary.stringForToken(TupleI.get(ts))));
                end;
             end;
          end  {bitlist2 check}
           else
           begin
                bset4.insert(i);
           end;
     end;
    if m>0 then
    begin
    {Storage optimization through "perfect split-out
    of several aggregated tuples"}
        k:=0;
        for i:=1 to m do
        begin
               if vSingleTable[i]<>0 then
               bset4.insert(VSingleTable[i])else
               begin
               newRelation.getTuple(i,tupleI);
```

```
                        k:=k+1;
                         if k<>i then
                        newRelation.putTuple(k,tupleI);
                    end;
             end;
               m:=k;
          end;
          if m>0 then       // m = NoofTuples
          begin
          {Finally convert Count or total to string or
token as appropriate}
          for i:=1 to m do
         newRelation.A.put(i,ts,dictionary.insertStringToken(IntToStr
                                   ( newRelation.A.get(i,ts))));
          newRelation.reSize(ts{tuplesize},m{no of aggregate tuples});
          relDiff:=TRelationDiff.create(newRelation,self,self.schema);
          relDiff.partRelation2.selectedTuples:= bset4;
          if (relDiff.partrelation1.noOfTuples>
              relDiff.partrelation2.noOfTuples*10)
          then
          begin
              aggregate:=relDiff.Deepcopy;
               relDiff.free;
{So few parent tuples remaining that it is better with a
simple Stored Relation instead of a Difference Relation}
          end else
```

```
                begin
                    aggregate:=relDiff;
                end;
            end else
            begin
                 newRelation.free;
                 aggregate:= self;
                 bset4.free;
            end;
            vAccessTable:=nil;   vHashNo:=nil; vSingleTable:=nil;
            bset1.free; bset2.free;bset3.free;
            tupleI.free; tupleJ.free;
        end else aggregate:=self;
end;
```

Appendix B

Source code: L-R data cube

```
{Author: Nikolaos Kotsis
 Synopsis: The following methods belong to Group-by and Cube-by objects
           which were used for the evaluation of the L-R approach}


constructor TGroupBy.initColumnNumber(aRelation:TRelation;
                    const aColumnNumber,aMeasureNumber:integer;
                    aggregate:TVariantFunc);
var newCol:integer;
begin
     projectedColumns:=TTuple.init(aRelation.tupleSize*2);
     relation:=aRelation;
     parentNoOfTuples:=relation.NoOfTuples;
     newSchema:=TStringlist.create;  firstGroupBy:=true;
     if aColumnNumber>aRelation.tuplesize then
        begin
```

```
          writeln('Error : ColumnNumber ',aColumnNumber,
                  ' exceeds tuple size ', aRelation.tuplesize);
          readln;
     end else
     begin
          newCol:=newSchema.add(aRelation.schema.strings[aColumnNumber-1]);
          projectedColumns.put(newCol+1,aColumnNumber);
          NoOfColumns:=newCol;
     end;
     if aMeasureNumber>aRelation.tuplesize then
     begin
          writeln('Error : MeasureColumnNumber ',aColumnNumber,
                  ' exceeds tuple size ', aRelation.tuplesize);
          readln;
     end else
     begin
          measureCol:=aMeasureNumber; aggregate1:=aggregate;
     end;
end;

constructor TGroupBy.initAllDimensionsMeasureName(aRelation:TRelation;
            const aMeasureName:string;  aggregate:TVariantFunc;
            firstGB:boolean;aRootRelation:TRelation);
var i,j,column:integer;
begin
     firstGroupBy:=firstGB;  {For use in CubeBy initialisation}
```

```
if firstGB then
RootRelation:=aRelation else RootRelation:=aRootRelation;
projectedColumns:=TTuple.init(aRelation.tupleSize{+1});
relation:=aRelation;
parentNoOfTuples:=relation.NoOfTuples;
newSchema:=TStringlist.create;
j:=0; i:=0;
while j<= (aRelation.schema.count-1) do
begin
     if aRelation.schema.Strings[j]<>aMeasureName then
     begin
        newSchema.add(aRelation.Schema.strings[j]);
        projectedColumns.put(i+1,j+1);
          i:=i+1;
     end;
j:=j+1;
end;
column:= aRelation.Schema.indexof(aMeasureName);
if column= (-1) then
begin writeln('Error - ',aMeasureName,' not found in schema');
      readln;
end else
begin
     measureCol:=column+1;   measureName:=aMeasureName;
     aggregate1:=aggregate;
end;
```

```
end;

constructor TGroupBy.initColumnName(aRelation:TRelation;
                        const aColumnName,aMeasureName:string;
                        aggregate:TVariantFunc);
var newCol,column:integer;
begin
    projectedColumns:=TTuple.init(aRelation.tupleSize*2);
    relation:=aRelation;
    parentNoOfTuples:=relation.NoOfTuples;
    newSchema:=TStringlist.create; firstGroupBy:=true;
    column:= aRelation.Schema.indexof(aColumnName);
    if column= (-1) then
    begin
        writeln('Error - ',acolumnName,' not found in schema');
        readln;
    end else
    begin
        newCol:= newschema.add(aColumnName)+1;
        column:=column+1;
        projectedColumns.put(newCol,column);
        NoOfColumns:=newCol;
    end;
    column:= aRelation.Schema.indexof(aMeasureName);
    if column= (-1) then
    begin
```

```
          writeln('Error - ',aMeasureName,' not found in schema');
          readln;
     end else
     begin
          measureCol:=column+1;   measureName:=aMeasureName;
          aggregate1:=aggregate;
     end;
end;

constructor TGroupBy.initMeasureName(aRelation:TRelation;
const aMeasureName:string; aggregate:TVariantFunc;
firstGB:Boolean; aRootRelation:TRelation);
var column:integer;
begin
     firstGroupBy:=firstGB;
     if FirstGB then RootRelation:=aRelation else RootRelation:= aRootRelation;
     projectedColumns:=TTuple.init(aRelation.tupleSize*2);
     relation:=aRelation;
     parentNoOfTuples:=relation.NoOfTuples;
     column:= aRelation.Schema.indexof(aMeasureName);
     if column= (-1) then
     begin
          writeln('Error - ',aMeasureName,' not found in schema');
          readln;
     end else
     begin
```

```
          measureCol:=column+1;   measureName:=aMeasureName;
          aggregate1:=aggregate; newSchema:=TStringList.create;
      end;
end;

function TGroupBy.hashschema:integer;
var i,sum,dsum:integer;
begin
{$Q-} {Overflowchecks off}
     dsum:=0; sum:=0;
     for i:= 0 to newSchema.count-1 do
     begin
         sum:=sum + hash(newSchema.strings[i]);
         dsum:=dsum+sum;
     end;
 if dsum<0 then dsum:= dsum shr 1;
     hashschema:=dsum;
end;

constructor TGroupby.initMeasureNamefromGB(aGB:TGroupBy;
const aMeasureName:string; aggregate:TVariantFunc;
firstGB:boolean; aRootRelation:TRelation);
var column:integer;
begin
     firstGroupBy:=firstGB;
     if firstGB then rootRelation:=aGB.Relation else rootRelation:=aRootRelation;
```

```
    projectedColumns:=TTuple.init(aGB.newSchema.count*2);
    relation:=aGB.newRelation;
    parentNoOfTuples:=aGB.relation.NoOfTuples;
    column:=aGB.newschema.indexof(aMeasureName);
    if column= (-1) then
    begin
        writeln('Error - ',aMeasureName,' not found in schema');
        readln;
    end else
    begin
        measureCol:=column+1;
        measureName:=aMeasureName;
        aggregate1:=aggregate;
        newSchema:=TStringList.create;
    end;
end;

procedure TGroupBy.exec1;
var newCol:integer;
begin
    newCol:= newschema.add(measureName)+1;
    projectedColumns.put(newCol,MeasureCol);
    NoOfColumns:=newCol;
    hNo:= self.hashschema;
    exec1Done:=true;
end;
```

```
destructor TGroupBy.free;
begin
     measureName:='';
     inherited free;
end;

procedure TGroupBy.freeRelation;
begin
     inherited freeRelation;
     exec2Done:=false;
end;

function TGroupBy.reducedAggregate:boolean;
var oldrelation:TRelationView;
begin
     if not exec1done then
        begin
           noOfColumns:= newschema.add(measureName)+1;
           projectedColumns.put(noOfColumns,MeasureCol);
        end;
      oldRelation:=TRelationView.create(relation, newSchema, projectedColumns);
        reducedAggregate:=oldRelation.reducedAggregate;
        oldRelation.free;
end;
```

```
procedure TGroupBy.exec;
var
oldRelation:TRelationView;
newCol:integer;
newRel:TRelation;
diffRel:TRelationDiff;
begin
if m<>-1 then
     begin
          if  not exec1done then
          begin
             newCol:= newschema.add(measureName)+1;
             projectedColumns.put(newCol,MeasureCol);
             NoOfColumns:=newCol;
             hNo:= self.hashschema;
          end;
          countBool:=@aggregate1=@count;
        oldRelation:=TRelationView.create(relation,newSchema,projectedColumns);
          oldRelation.measurecol:=oldRelation.tupleSize;
          if firstGroupBy and countBool then oldRelation.setCountIs1;
          parentNoOfTuples:= relation.NoOfTuples;
          newRel:=oldRelation.aggregate(firstGroupBy,countBool,keyPresent);
          resultRelation:=newRel;
          newRelation:=newRel;
          m:=-1;
     end;
```

```
end;

procedure TGroupBy.setKeyPresent;
begin
     keyPresent:=true;
end;

function TGroupBy.tupleCost:integer;
begin
     tupleCost:=parentNoOfTuples; { + newRelation.noOfTuples;}
end;

function TGroupBy.estCost:integer;
begin
     if storedEstCost<=0 then storedEstCost:=self.tuplecost;
     estCost:=storedEstCost;
end;

{start of Cube}

procedure TCubeBy.display;
var  i:integer;
begin
     writeln;
     writeln('Number of GroupBys in CubeBy = ',NoOfGroupBys);
     for i:=1 to NoOfGroupBys do
```

```
    begin
        writeln('Tuple cost = ', vGroupBy[i].tupleCost);
        readln;
    end;
end;

function TCubeBy.getValue(i:integer):TRelation;
begin
    getValue:=vGroupBy[i].value ;
end;

function TCubeBy.getValueSchema(s:TStringList):TRelation;
var
found:boolean; i:integer;
begin
    i:=1;
    found :=false;
    while (i<=noOfGroupBys) and (not found) do
    begin
        found:=s.equals(self.getValue(i).Schema);
        i:=i+1;
    end;
    i:=i-1;
    if (not found) then
    getValueSchema:=nil
    else
```

```
     getValueSchema:=self.getValue(i);
end;
destructor TCubeBy.free;
var i:integer;
begin
     vGroupBy:=nil;  vGBAccessTable:=nil;
     for i:=1 to noOfCompGBs do
     compGBList[i].free;
     compGBList:=nil;
     for i:=1 to noOfKeys do
     keyList[i].free;
     keyList:=nil;
     {inherited free;}
  end;

procedure TCubeBy.freeAllGroupBys;
var i:integer;
begin
     for i:=1 to noOfGroupBys do
     begin
          vGroupBy[i].freeRelation;
        vGroupBy[i].free;
     end;
     self.free;
end;
procedure TCubeBy.fetchAllGroupBys;
```

```
var i:integer;
begin
     for i:=1 to noOfGroupBys do
     begin
          vGroupBy[i].copyOfValue.free;
     end;
end;

function TCubeBy.mustBeSmallerThanBaseRelation(aDimensionSchema:
TstringList):boolean;
var i:integer; n:integer;
begin
     n:=1;
     i:=1;
     while (n<=baseRelation.NoofTuples) and (i<=aDimensionSchema.count)  do
     begin
          n:=n*baseRelation.cardinality(aDimensionSchema.strings[i-1]);
          i:=i+1;
     end;
     mustBesmallerThanBaseRelation:=n<baseRelation.NoOfTuples;
end;

function TCubeBy.reducedAggregate(anAggregateSchema:TstringList): boolean;
var
aGB:TGroupBy;
i:integer;
```

```
begin
     aGB:=TGroupBy.initMeasurename(baseRelation,measureName,aggregatef,
false,nil);
     for i:=0 to anAggregateSchema.count-1 do
     if measureName<>anAggregateSchema.strings[i] then
     aGB.addColumnname(anAggregateSchema.strings[i]);
     reducedaggregate:=aGB.reducedAggregate;
     aGB.free;
end;

function TCubeBy.Containskey(y:Tstringlist; x:Tstringlist) : boolean;
var
i,l,size:integer;
contain:boolean;
begin
     contain:=false;
     l:=0;
     size:=y.count;
     for i:=0 to size-1 do
     begin
          if x.indexof(y.strings[i])<> -1  then
          l:=l+1;
     end;
     if l=size then
     contain:=true
     else
```

```
     Containskey:=false;
     Containskey:=contain;
end;

function TCubeBy.ContainsAnykey(x:Tstringlist) : boolean;
var
i:integer;
keyfound:boolean;
begin
   i:=1;
   keyfound:=false;
   while (i<=noOfKeys) and (not keyfound) do
   begin
        keyfound:=containskey(Keylist[i],x);
        i:=i+1;
   end;
   Containsanykey:=keyfound;
end;

function TCubeBy.power(x:integer):integer;
var
i:integer;
y,k:integer;
begin
     k:=1;
     y:=2;
```

```
    for i:=1 to x do
    begin
        k:=y*k;
    end;
    power:=k;
end;

procedure TCubeBy.makeKeyList(aRelation:TRelation;const aMeasureName:string;
aggregate:TVariantFunc);
var
posin,line:integer;f1,f2:textfile;
j,i,n:integer;
noOfCombinations:integer;
mask:integer;
dimensionSchema:TStringList;
newAggregateSchema:TstringList;
begin
    assignfile(f1,'c:\GBList.txt');
    assignfile(f2,'c:\Klist.txt');
    Rewrite(f1);
    Rewrite(f2);
    {Obtain the dimensions-only schema, leaving out the measure of interest}
    dimensionSchema:=TStringList.Create;
    for i:=0 to  aRelation.schema.count-1 do
    if aMeasureName <> aRelation.schema.strings[i] then
    dimensionSchema.add(aRelation.schema.strings[i]);
```

```
    noOfCombinations:=power(dimensionschema.count);
    setLength (keyList,noOfCombinations+1);
setLength(compGBList,noOfCombinations+1);
    noOfCompGBs:=0; noOfKeys:=0;
    for i:=1 to noOfCombinations do
    begin
        {Form the schema required by the current ith. combination}
        newAggregateSchema:=TStringList.create;
        for j:=0 to dimensionSchema.count-1 do
        begin
            mask:=1 shl j;
            if (i and mask)<>0  then
  newAggregateSchema.add(dimensionSchema.strings[j]);
        end;
        {Decide what to do with it}
        if mustBesmallerThanBaseRelation(newAggregateSchema) then
        begin
            {It can't be a key so add it to the compGBList}
            noOfCompGBs:=noOfCompGBs+1;
            compGBList[noOfCompGBs] :=newAggregateSchema;
        end
        else if containsanykey(newAggregateSchema) then
        begin
            newAggregateSchema.free;
            {minimum number of minimal keys}
        end
```

```
          else if reducedAggregate(newAggregateSchema) then
          begin
               {Add it to the compGBList}
               noOfCompGBs:=noOfCompGBs+1;
               compGBList[noOfCompGBs] :=newAggregateSchema;
          end else
          begin
               { This is a new key: add it to the keyList}
               noOfKeys:=noOfKeys+1;
               keyList[noOfKeys]:=newAggregateSchema;
          end;
      end;
      dimensionSchema.free;
end;

procedure TCubeBy.initMeasureName2(aRelation:TRelation;
                              const aMeasureName:string;
                              aggregate:TVariantFunc);
var start,jPos:integer;

function sameGB(gB1,gB2:TGroupBy):boolean;
var i :integer; same:boolean;
begin
     i:=0;
     same:=(gb1.hNo=gB2.hNo) and(gB1.newSchema.count=gB2.newSchema.count);
     while same and (i<gB1.newSchema.count) do
```

```
     begin
          same:= gb1.newSchema.strings[i]=gB2.newSchema.strings[i];
          if same then  i:=i+1;
     end;
     sameGB:=same;
end;


function alreadyPresent(aGB:TGroupBy):boolean;
          var found:boolean; i,j,step,c:integer;
          begin
               found:=false;   {hno=hashSchema(aGB.newSchema);  }
               j:= aGB.hNo mod (maxNoOfGroupBys*3);
   step:= aGB.hNo mod 13 +1; c:=0;
               i:=vGBAccessTable[j];
               while (i<>0{NoOfGroupBys}) and (not found) do
               begin
                found:=(aGB.hNo=vGroupBy[i].hNo)and sameGB(aGB,vGroupBy[i]);
                    if not found then
                    begin
                     if c>5 then step:=1;
                     j:=(j+step)mod (maxNoOfGroupbys*3);
                     c:=c+1;
                        i:=vGBAccessTable[j];
                    end;
                    {i:=i+1; }
               end;
```

```
                if found then
                begin
                    {i:=i-1;  }
                     if (i>=start) and (aGB.estCost<vGroupBy[i].estCost) then
                     begin
                      vGroupBy[i].free;
                          vGroupBy[i]:=aGB;
                     end else aGB.free;
                end;
                jpos:=j;
                alreadyPresent:=found;
           end;

procedure makeGBs(aGB:TGroupBy);
    var
ts,k,j,ii:integer;
gB:TGroupBy;
r:TRelation;
keyPresent:boolean;
    testSchema:TstringList;
    begin
         ts:=aGB.NoOfColumns;
         for k:= 1 to (ts-1) do
        begin
              r:=aGB.value;
              TestSchema:=TStringList.Create;
```

```
for j:= 1 to (ts-1) do
 if (j<>k)and(r.schema.strings[j-1]<>aMeasureName) then
     testSchema.add(aGB.newschema.strings[j-1]);
     keypresent:=containsAnyKey(testSchema);
     {keypresent:=false;}
     if keypresent then
     begin
          gb:=TGroupBy.initMeasureNamefromGB(aGB,aMeasureName,
                             aggregate,FALSE,aGB.rootRelation);
         gb.setKeyPresent;
     end
     else
 gB:=TGroupBy.initMeasureNamefromGB(aGB,aMeasureName,
        aggregate,FALSE,aGB.rootRelation);
     for j:=1 to TestSchema.count do
     gB.addColumnName(TestSchema.strings[j-1]);
     testSchema.free;
     gb.exec1;
     if not alreadyPresent(gB) then
     begin
          ii:=ii+1;
          vGroupBy[ii]:=gB;
          NoOfGroupBys:=NoOfGroupBys+1;
             vGBAccessTable[jPos]:=NoOfGroupBys;
    end;
end;
```

```
   end;
   var
   ts:integer;
   gB:TGroupBy;
   r:Trelation;
   k3:integer;
   f1:textfile;
   begin
        assignfile(f1,'c:\outGBs.txt');
        rewrite(f1);
        NoOfGroupBys:=1;  start:=1;
        { Allocate max number GroupBys which might be needed}
        ts:=aRelation.tupleSize;
        ts := 2 shl ts;   maxNoOfGroupBys:=ts;
        setlength(vGroupBy,ts); setlength(vGBAccessTable,ts*3+1);
        gB:=TGroupBy.initAllDimensionsMeasureName(aRelation,aMeasureName,
  aggregate,true,nil);
        gb.exec1;
        vGroupBy[NoOfGroupBys]:= gB;
        while start<=NoOfGroupBys do
        begin
            start:=start+1;
            makeGBs(vGroupBy[start-1]);
end;
        setlength(vGroupBy,start); {Reduce to required length}
        closefile(f1);
```

```
end;

constructor TCubeBy.initMeasureName(aRelation:TRelation;
const aMeasureName:string;aggregate:TVariantFunc);
var
timer1,timer2,timer3,timer4:comp;
existingSelectedIndex:boolean;
begin
     if aRelation.selectedindex then
     existingselectedindex:=true else
     aRelation.makeSelectedIndex;
     baseRelation:=aRelation;
     MeasureName:=aMeasureName;
     aggregatef:=aggregate;
     self.makeKeyList(aRelation,aMeasureName,aggregate);
     self.initMeasureName2(aRelation,aMeasureName,aggregate);
     if not existingselectedindex then
     aRelation.freeSelectedIndex;
end;

function TCubeBy.tupleCost:integer;
var  i:integer;   cost:integer;
begin
     cost:=0;
     for i:=1 to NoOfGroupBys do
     begin
```

```
        cost:=cost+ vGroupBy[i].tupleCost;
    end;
    tupleCost:=cost;
end;
end.
```

Appendix C

Analytical Results

C.1 Computing the Data Cube

The timings shown are the total time, consisting of the time to select the non-redundant views utilising the Key-algorithm and the time required to compute the non-redundant data cube. These timings are compared to those taken for the conventional data cube approach. All figures in this section show the results in a logarithmic scale and every figure has an associated table showing the results analytically.

The synthetic datasets

Figure C.1 and Table C.1 show time performance of the L-R approach using the 600K TPC-D dataset. Figure C.1 illustrates that the Totally-Redundant approach (g-equivalent) is faster than the conventional data cube and its performance depends on the dimensionality of the dataset. The combined version is slower than the Totally-Redundant version but is still faster than the conventional one. Table C.1 gives the results in detail. The results using the 60K dataset TPC-D are shown in Figure C.2 and Table C.2. The

Totally-Redundant approach is again faster than the conventional approach and increases its performance after the five-dimension data cube has been used. The combined version is slower until it reaches the ten-dimension data cube. In the ten-dimension data cube, the combined approach is faster than the conventional data cube. Figure C.3 and Table C.3 show the experimental results taken from 6K dataset TPC-D. The Totally-Redundant approach is slower until the five-dimension data cube is used, but after the six-dimension data cube it its performance increases and after this point, remains faster than the conventional data cube. The combined approach is slower than both the Totally-Redundant and conventional approaches but matches the performance of the conventional approach with the ten-dimension data cube. Figure C.4 and Table C.4 show the results using the hotel dataset. Similar conclusions can be drawn from this test. The Totally-Redundant approach is faster than both the conventional and the combined approach after the five-dimension data cube. The combined approach is slower than both the conventional and Totally-Redundant approaches but it increases its performance in the nine-dimension data cube where it becomes faster than the conventional data cube.

The real datasets

In both the weather and adult datasets, the performance time of the L-R approach is similar. Figure C.5 and Table C.5 show the results using the weather dataset and Figure C.6 and Table C.6 show the results using the adult dataset. These results, when compared to the synthetic datasets, demonstrate that the performance of the Totally-Redundant approach is affected negatively. The conventional data cube is faster in all cases and only in the weather dataset (see Figure C.5) is it close to the combined approach. This arises from the non-existence of keys in these real datasets, as was discussed in section 3.5.2. The time overhead, in the Totally-Redundant approach, is time expended by the Key algorithm without benefit.

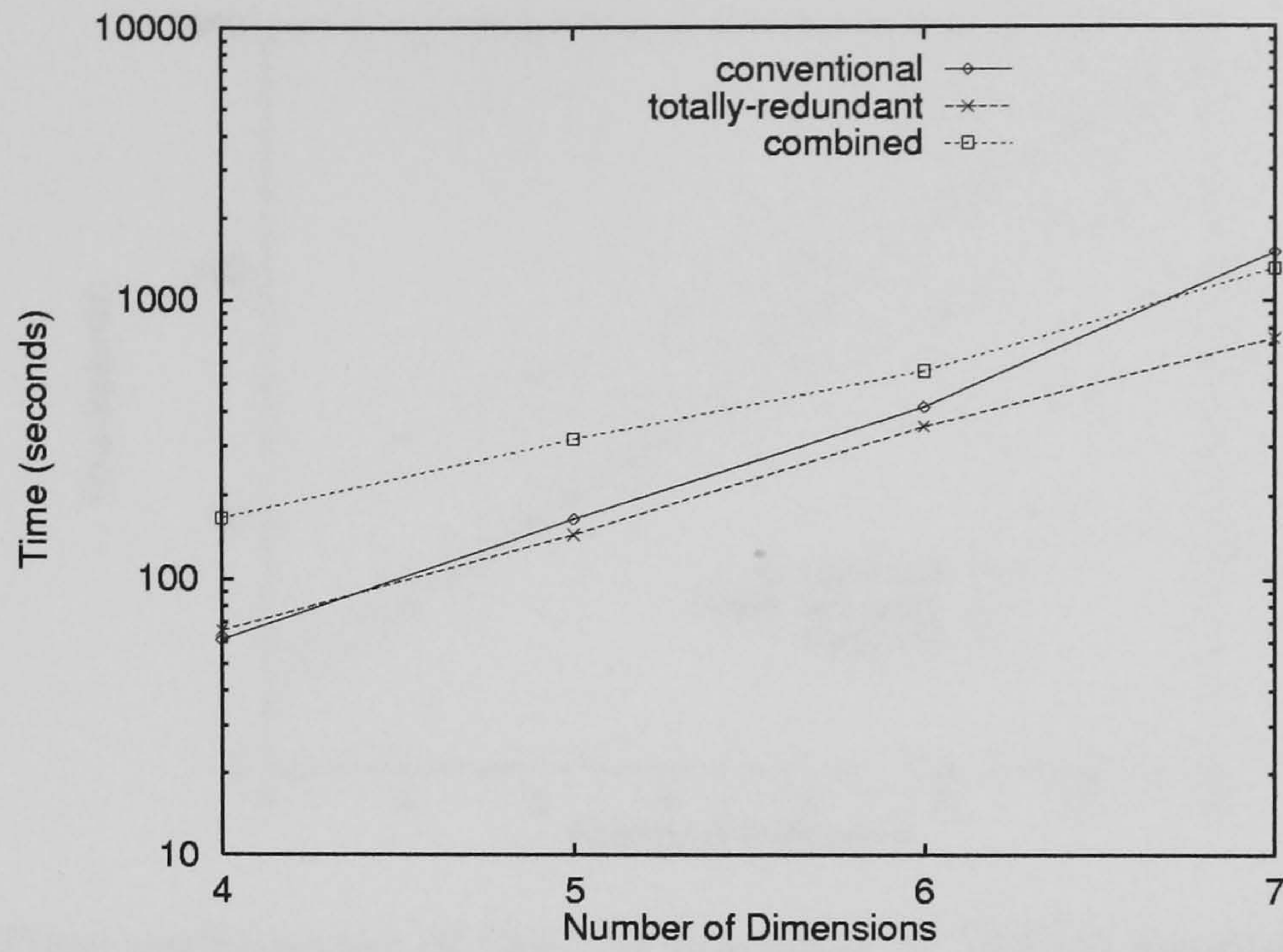

Figure C.1: Time performance of the L-R approach in TPC-D Lineitem Table (600K)

	Time(sec)		
Dimensions	Conventional	Totally-Redundant	Combined
4	60.74	65.6	165
5	164	144.5	317
6	415	353.3	558
7	1500	734.3	1309

Table C.1: Time performance of the L-R approach in TPC-D Lineitem Table (600K)

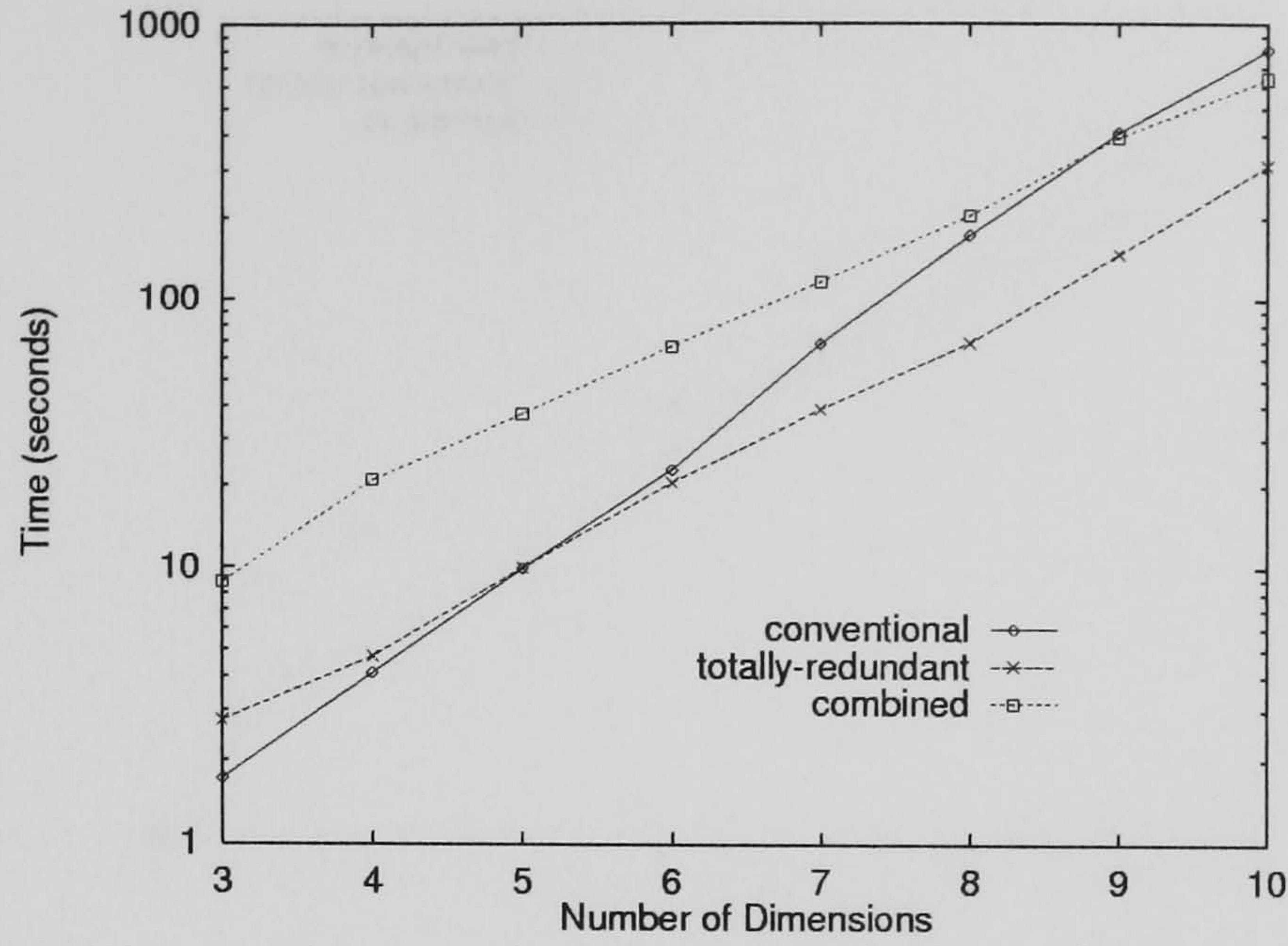

Figure C.2: Time performance of the L-R approach in TPC-D Lineitem Table (60K)

	Time(seconds)		
Dimensions	Conventional	Totally-Redundant	Combined
3	1.72	2.77	8.8
4	4.11	4.71	20.9
5	9.89	9.95	37
6	22.98	20.54	66.32
7	68.37	38.87	116.05
8	172.7	68.98	204.7
9	410	147	394.8
10	810	312	641

Table C.2: Time performance of the L-R approach in TPC-D Lineitem Table (60K)

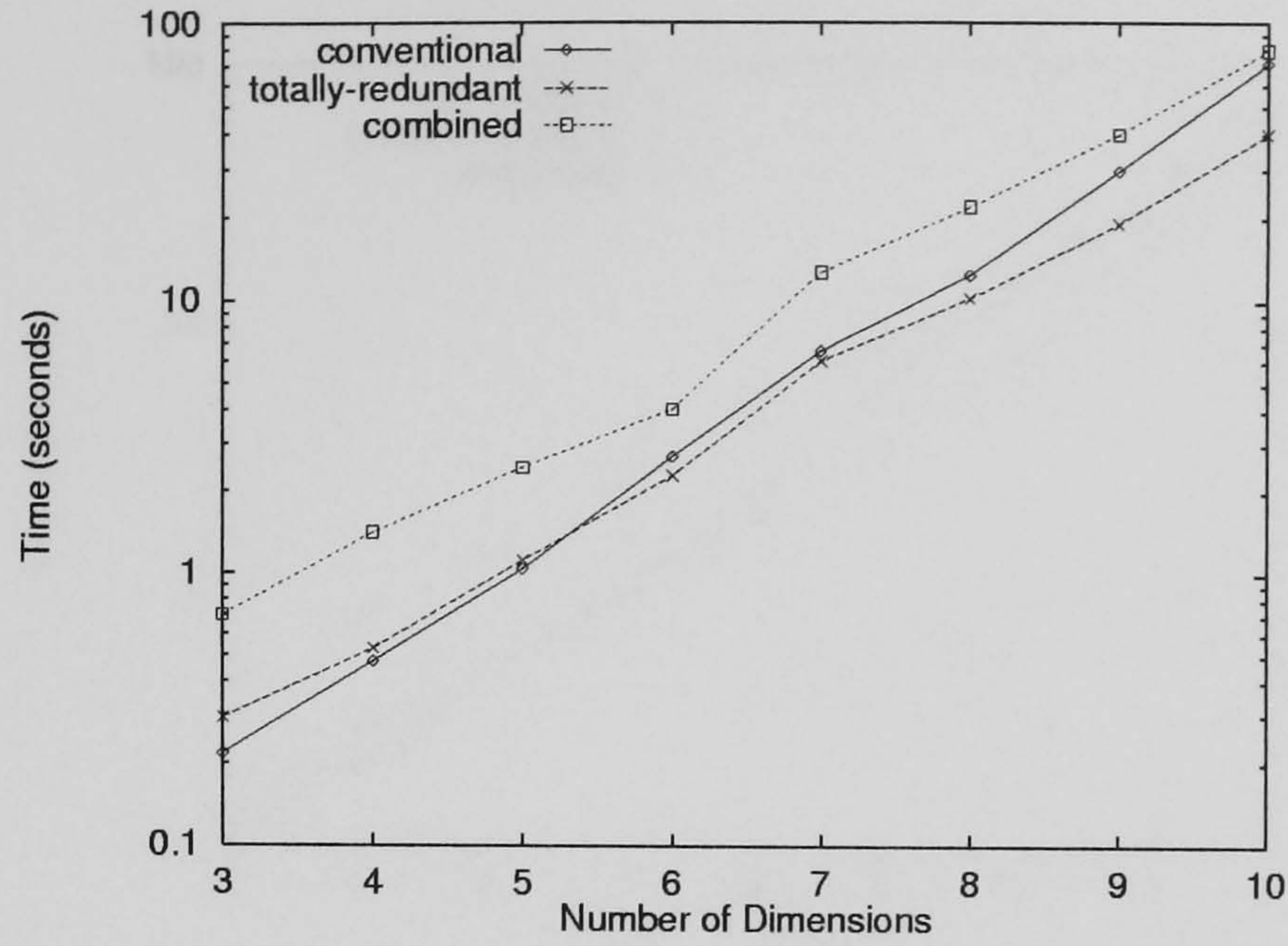

Figure C.3: Time performance of the L-R approach in TPC-D Lineitem Table (6K)

	Time(seconds)		
Dimensions	Conventional	Totally-Redundant	Combined
3	0.21	0.29	0.83
4	0.0.47	0.53	1.41
5	1.04	1.13	2.46
6	2.270	2.31	4.04
7	6.61	6.09	12.8
8	12.55	10.30	22.1
9	29.68	19.2	40.15
10	71.89	40.24	79.89
11	167	91.82	163

Table C.3: Time performance of the L-R approach in TPC-D Lineitem Table (6K)

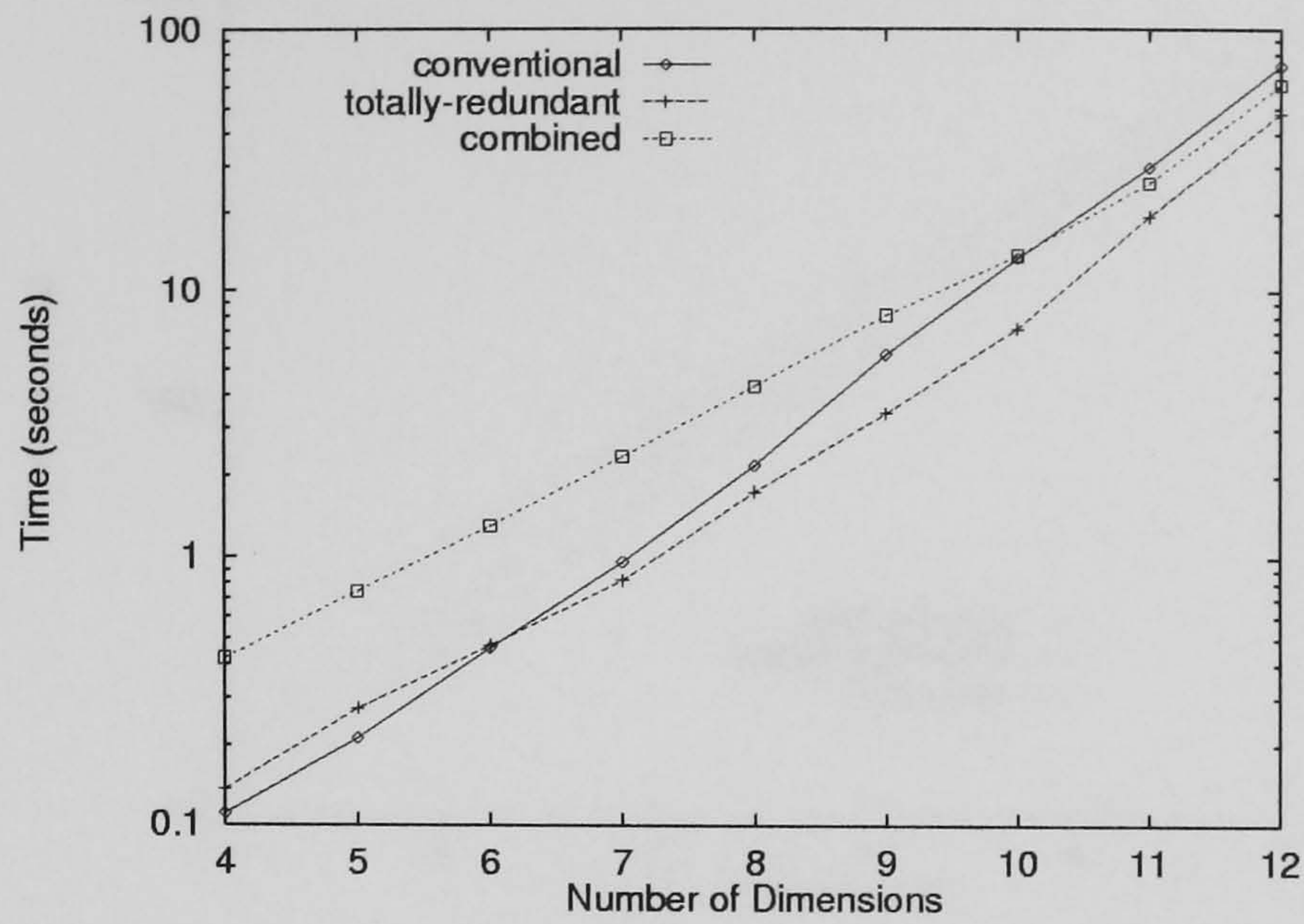

Figure C.4: Time performance of the L-R approach in Hotel dataset

	Time(seconds)		
Dimensions	Conventional	Totally-Redundant	Combined
4	0.11	0.13	0.42
5	0.21	0.27	0.745
6	0.46	0.47	1.30
7	0.96	0.82	2.36
8	2.20	1.75	4.35
9	5.72	3.45	8.10
10	13.43	7.2	13.76
11	29.90	19.39	26.02
12	72.78	48.19	61.58

Table C.4: Time performance of the L-R approach in Hotel dataset

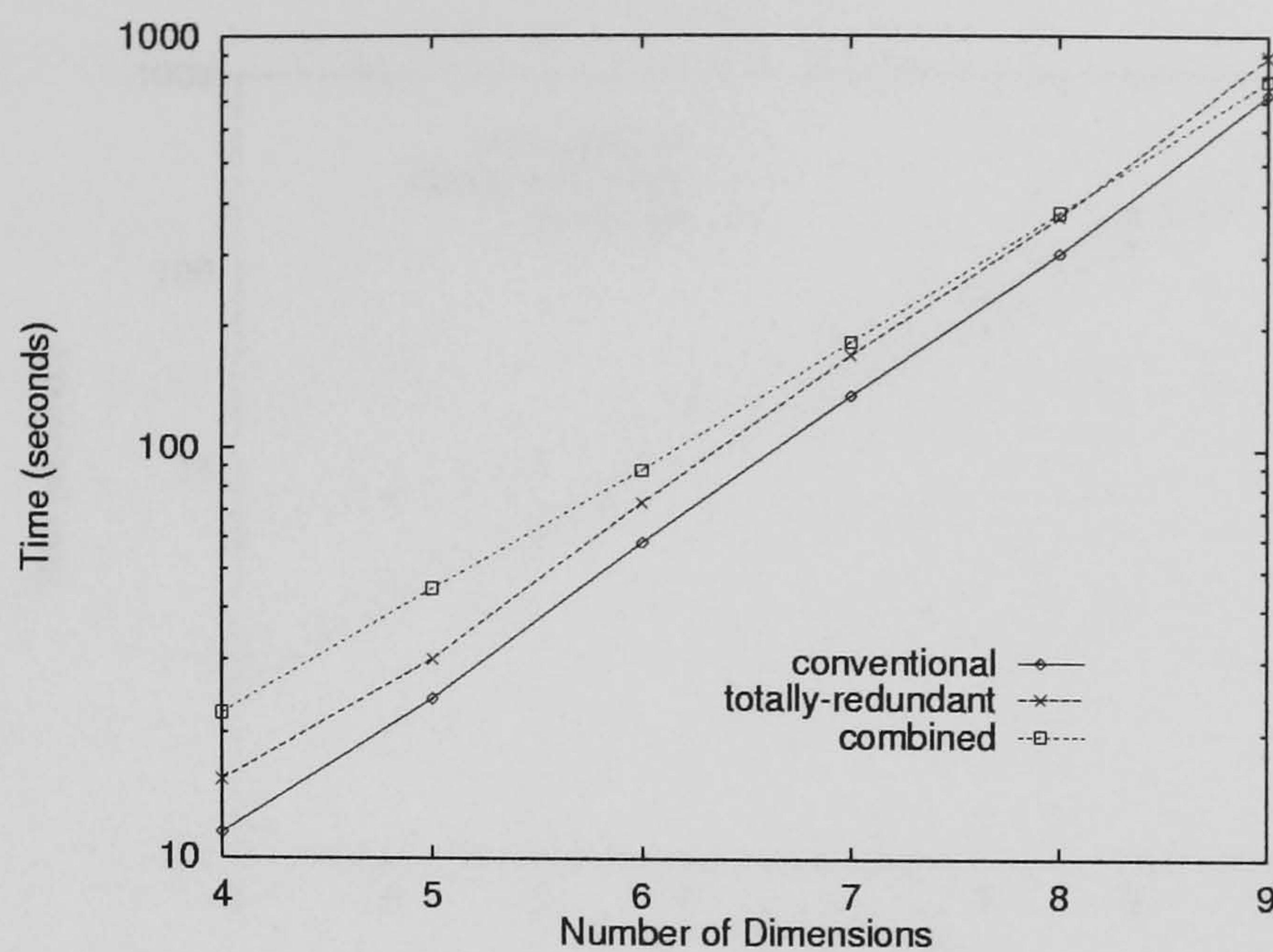

Figure C.5: Time performance of the L-R approach in the Weather dataset

	Time(seconds)		
Dimensions	Conventional	Totally-Redundant	Combined
4	11.48	15.4	22.3
5	24.18	30.21	44.7
6	58.33	73.16	88.31
7	135.7	171.11	184.34
8	303.8	372.9	381.6
9	729	896.5	785

Table C.5: Time performance of the L-R approach in the Weather dataset

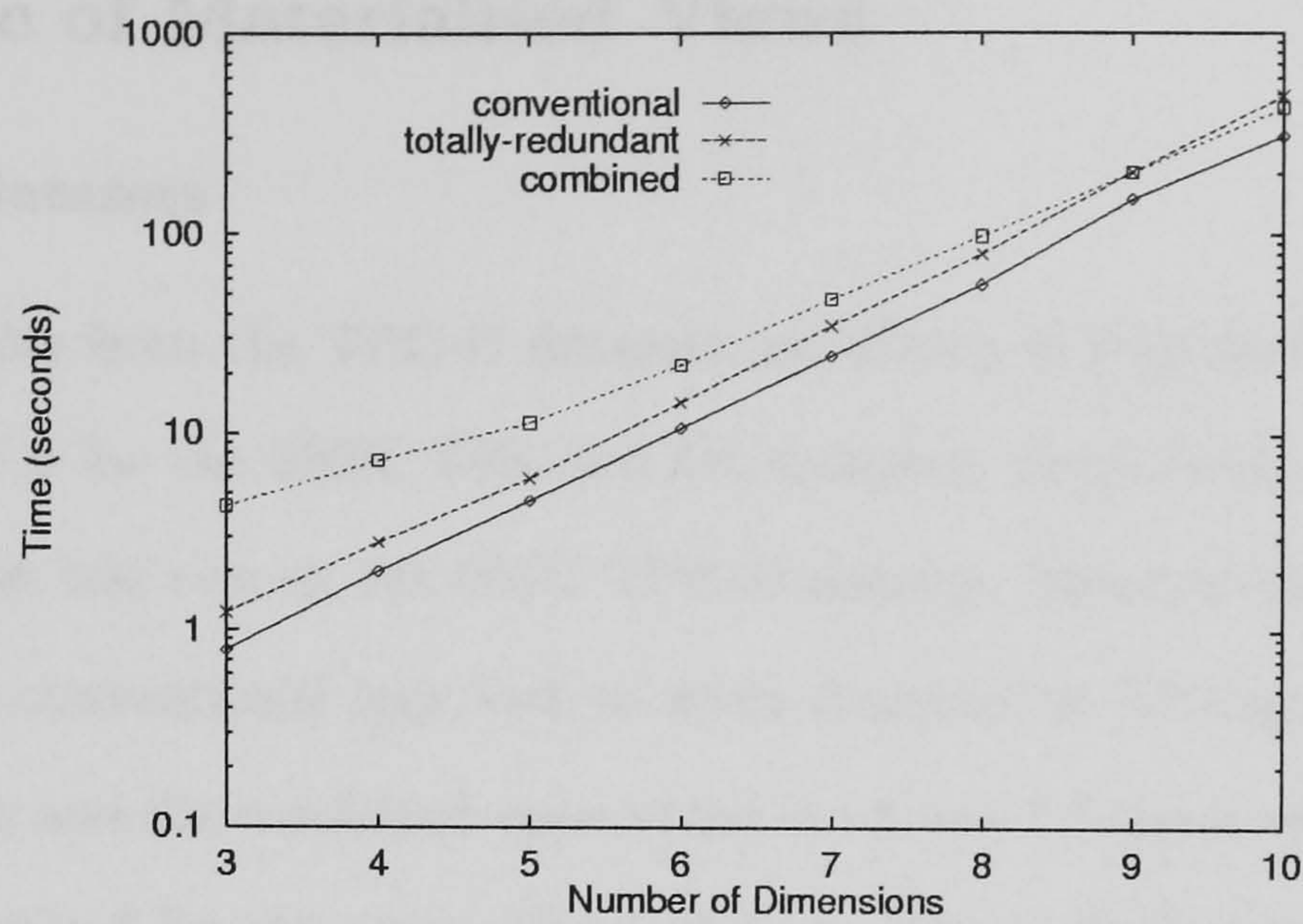

Figure C.6: Time performance of the L-R approach in the Adult dataset

	Time (seconds)		
Dimensions	conventional	Totally-Redundant	Combined
3	0.79	1.23	4.27
4	1.99	2.78	7.33
5	4.55	5.93	11.26
6	10.61	14.27	22.06
7	24.61	34.71	47.27
8	56.48	80.15	98.47
9	150	204.51	203
10	307.57	486	424

Table C.6: Time performance of the L-R approach in the Adult dataset

C.2 Storage of Materialised Views

The Synthetic datasets

Experimental results from the TPC-D datasets are shown in Figures C.7, C.8, C.9 and Tables C.7, C.8, C.9 for the 600K, 60K and 6K datasets, respectively. Figure C.7 shows the results from the test run on the 600K TPC-D dataset. Memory limitations restricted the tests using the conventional approach to seven dimensions. The space required for the Totally-Redundant and the combined approaches is 1.8 and 7.7 times smaller, respectively, than the space required for the conventional storage. These results are shown in detail in Table C.7.

Figure C.8 illustrates the space savings of the L-R approach in ten dimensions for the TPC-D 60K dataset. Due to the higher dimensionality, the savings are higher than those achieved using the 600K dataset. Here the space requirements are 3.6 (Totally-Redundant) and 26.7 (combined) times smaller than the conventional approach. The results can be seen in Table C.8.

Similar results were recorded using the 6K dataset and are shown in Figure C.9 and in detail in Table C.9. These results show that space requirements are 3.26 (Totally-Redundant) and 24.5 (combined) times smaller than the conventional method.

The hotel dataset has produced similar results to the TPC-D data. This was expected since it is a uniform dataset. These results are shown in Figure C.10 and Table C.10. Experiments were run with up to twelve dimensions and the indicated savings are larger than the TPC-D datasets; space savings are 7.66 (Totally-Redundant) and 38.1 (combined) times smaller than the conventional storage method.

The Real datasets

Results from the weather dataset are shown in Figure C.11 and Table C.11. One difference between this dataset and the synthetic ones, is that the Totally-Redundant method does not perform as well, in terms of space savings. This results from the small number of keys found by the Key-algorithm. Note however, that the savings increase with the number of dimensions and that after the ninth-dimension cube, the savings are more apparent than those achieved in a smaller number of dimensions. The combined method is insensitive to data skewness and performs equally as well in terms of space savings, as the synthetic datasets discussed previously. The space requirements were 1.1 (Totally-Redundant) and 15.3 (combined) times smaller than the conventional approach in nine dimensions.

For the adult dataset the test results are shown in Figure C.12 and Table C.12. The performance of the Totally-Redundant approach is also poorer in this dataset than in that of the uniform (synthetic) datasets. The combined version, however, is insensitive to data skewness and dramatically reduces space requirements in the storage of materialised views. Space savings are up to 30 times greater than for the conventional approach. This is shown in Table C.12. With the same number of dimensions, the g-equivalent approach is at least 2.2 times more economical than the conventional approach in ten dimensions. Note the sudden increase in space savings of the g-equivalent approach in the transition from 9 to 10 dimensions. This change occurred because the observational keys found previously affect more views in ten dimensions than in any smaller dimensionality.

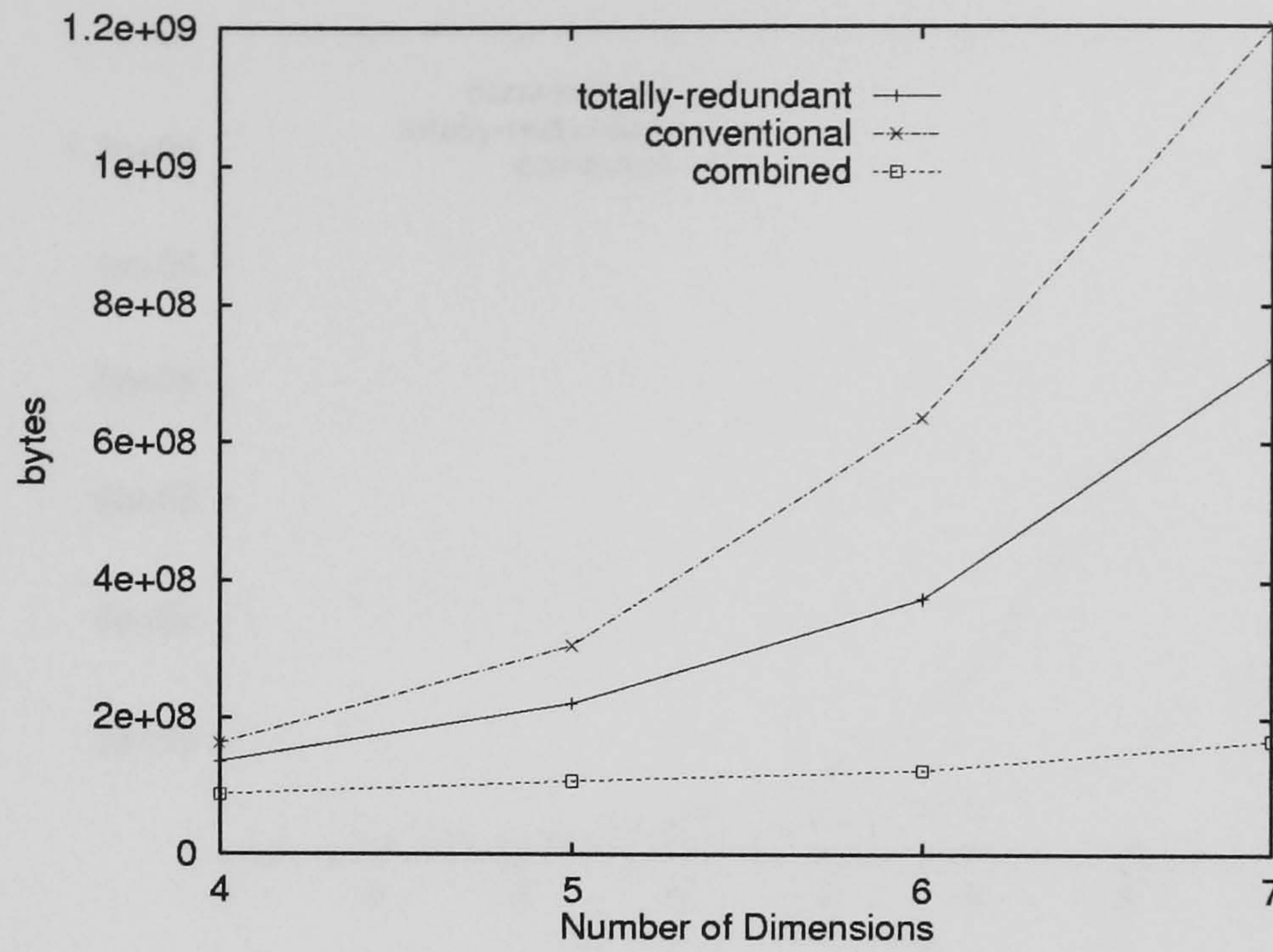

Figure C.7: Space savings of the L-R approach in TPC-D Lineitem Table (600K)

	Memory allocated (bytes)		
Dimensions	Conventional	Totally-Redundant	Combined
4	162,723,092	136,298,576	88,434,024
5	305,755,268	221,675,992	108,385,160
6	636,167,484	374,318,332	124,770,880
7	1,300,000,000	722,573,540	167,893,536

Table C.7: Space savings of the L-R approach in TPC-D Lineitem Table (600K)

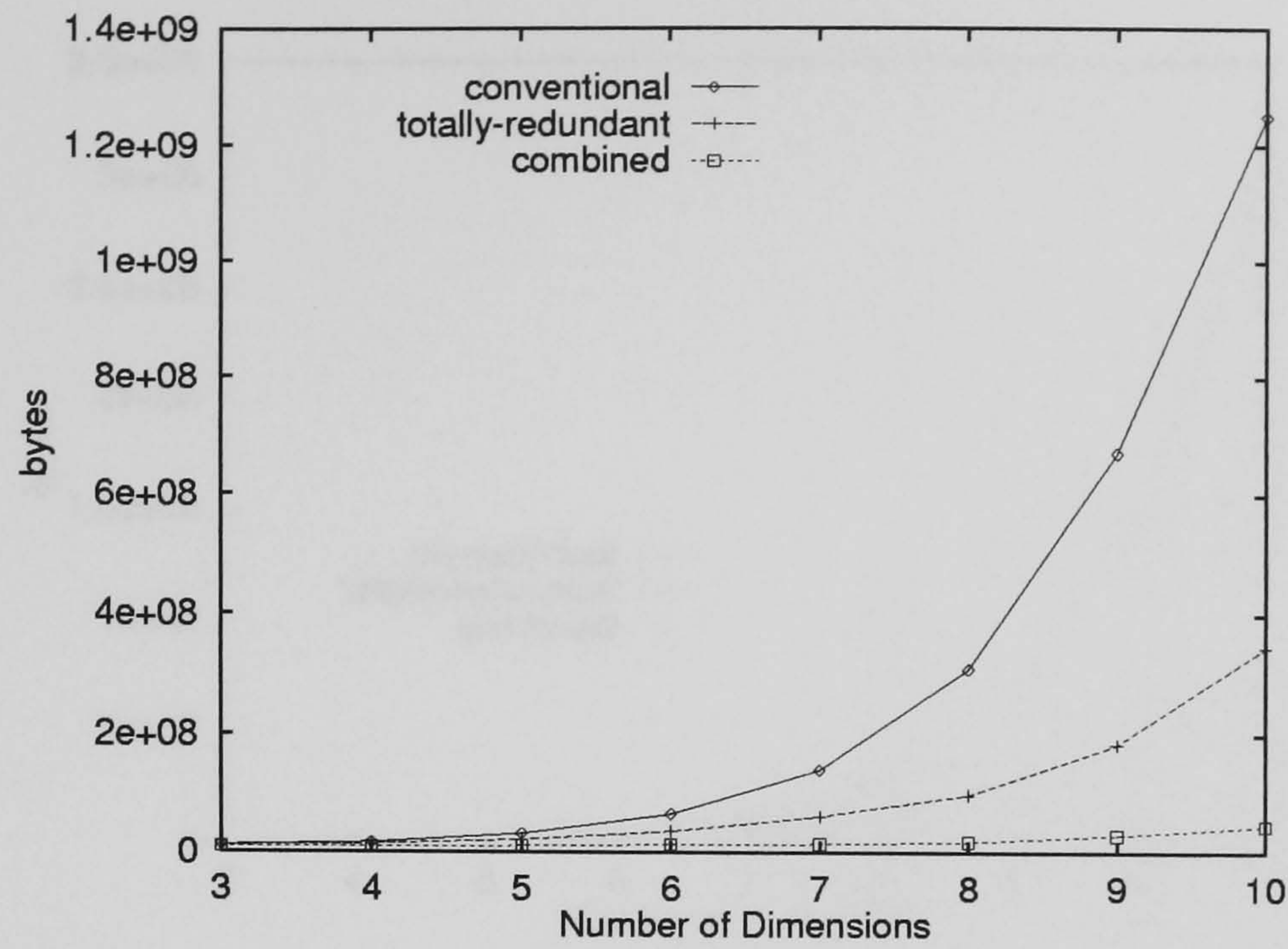

Figure C.8: Space savings of the L-R approach in TPC-D Lineitem Table (60K)

	Memory allocated (bytes)		
Dimensions	Conventional	Totally-Redundant	Combined
3	12,668,100	12,656,196	9,787,756
4	17,460,120	14,812,988	9,816,276
5	32,468,468	22,118,688	11,506,872
6	66,264,468	36,898,344	13,761,780
7	142,350,724	62,433,700	15,166,048
8	310,612,908	99,743,252	18,554,036
9	672,372,452	185,156,856	30,622,664
10	1,250,360,420	347,195,728	46,746,652

Table C.8: Space savings of the L-R approach in TPC-D Lineitem Table (60K)

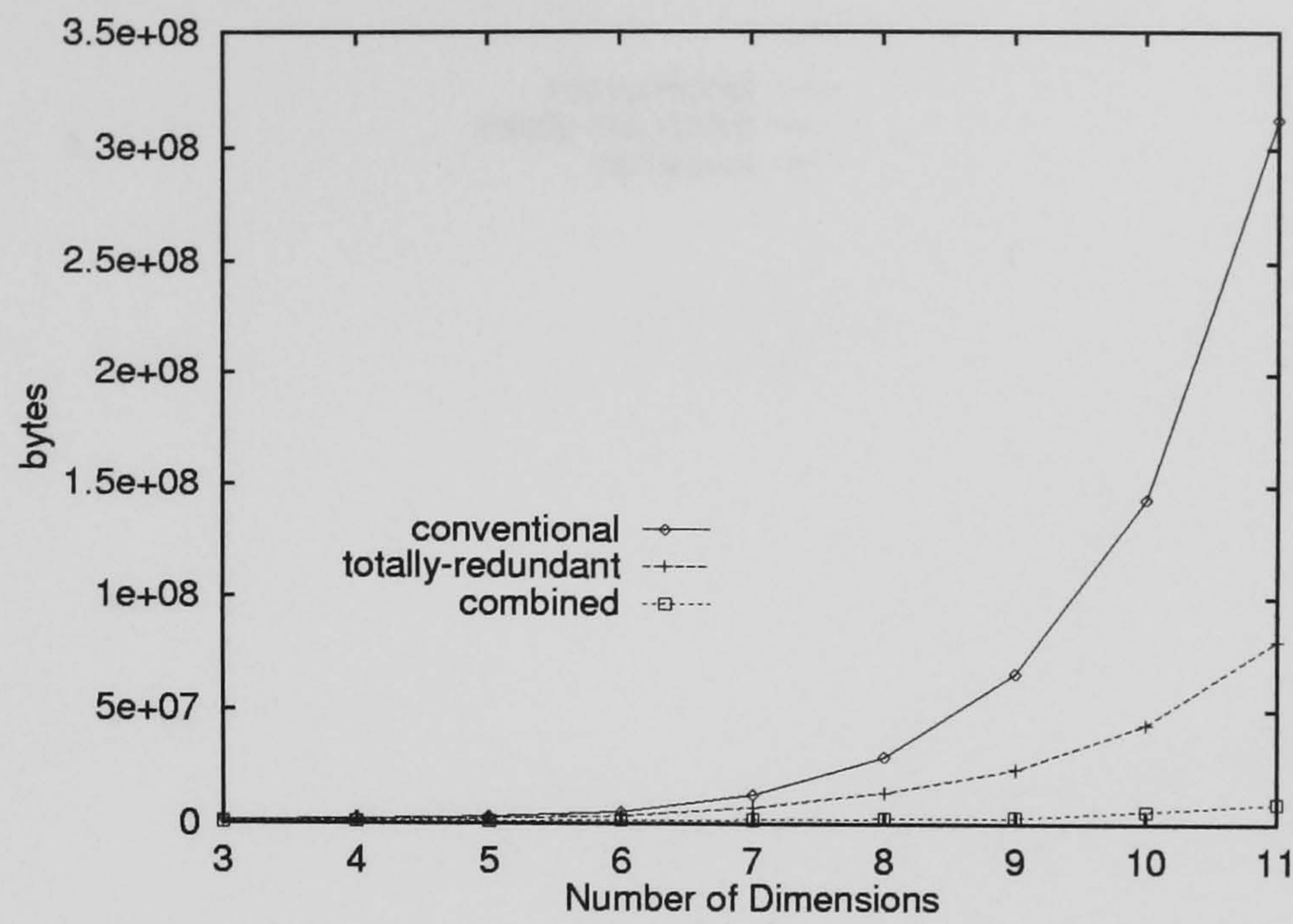

Figure C.9: Space savings of the L-R approach in TPC-D Lineitem Table (6K)

	Time(sec)		
Dimensions	Conventional	Totally-Redundant	Combined
3	1,644,740	1,645,300	943,567
4	2,121,536	1,857,852	1,138,956
5	3,130,460	2,410,580	1,310,004
6	5,302,204	3,550,080	1,549,676
7	12,760,780	7,356,324	20,232,605
8	29,469,580	13,804,896	2,490,540
9	66,209,240	24,099,268	2,914,036
10	143,761,944	44,036,512	5,845,800
11	312,391,104	80,910,700	9,133,652

Table C.9: Space savings of the L-R approach in TPC-D Lineitem Table (6K)

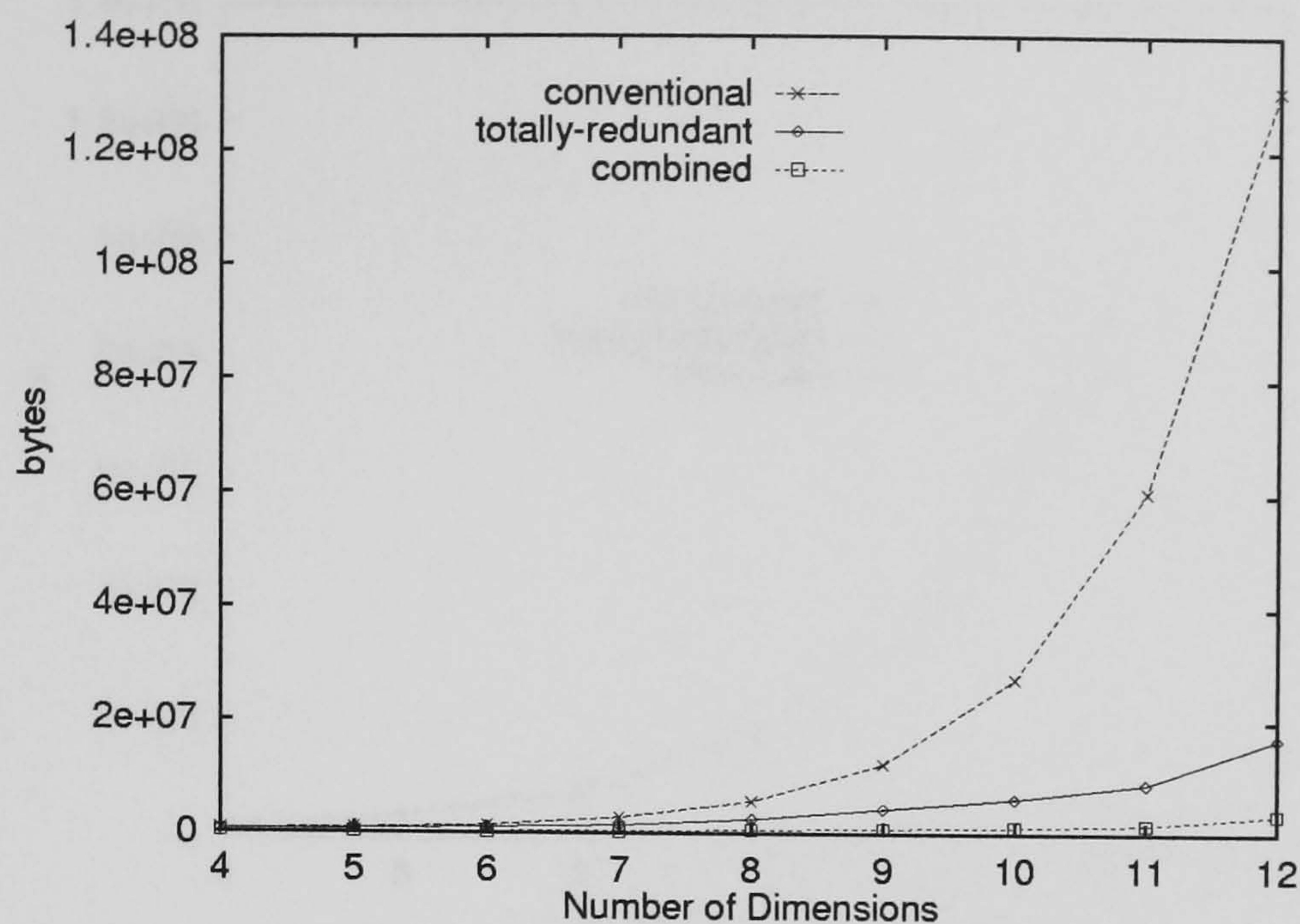

Figure C.10: Space savings of the L-R approach in the hotel dataset

	Memory allocated (bytes)		
Dimensions	Conventional	Totally-Redundant	Combined
4	656,044	627,044	253,240
5	877,604	770,436	305,728
6	1,423,672	1,081,468	376,152
7	2,855,160	1,442,908	405,132
8	5,734,132	2,546,156	662,560
9	12,428,092	4,440,444	936,060
10	27,569,012	6,236,620	1,159,964
11	60,352,868	8,922,480	1,638,860
12	130,535,816	17,019,420	3,423,164

Table C.10: Space savings of the L-R approach in the hotel dataset

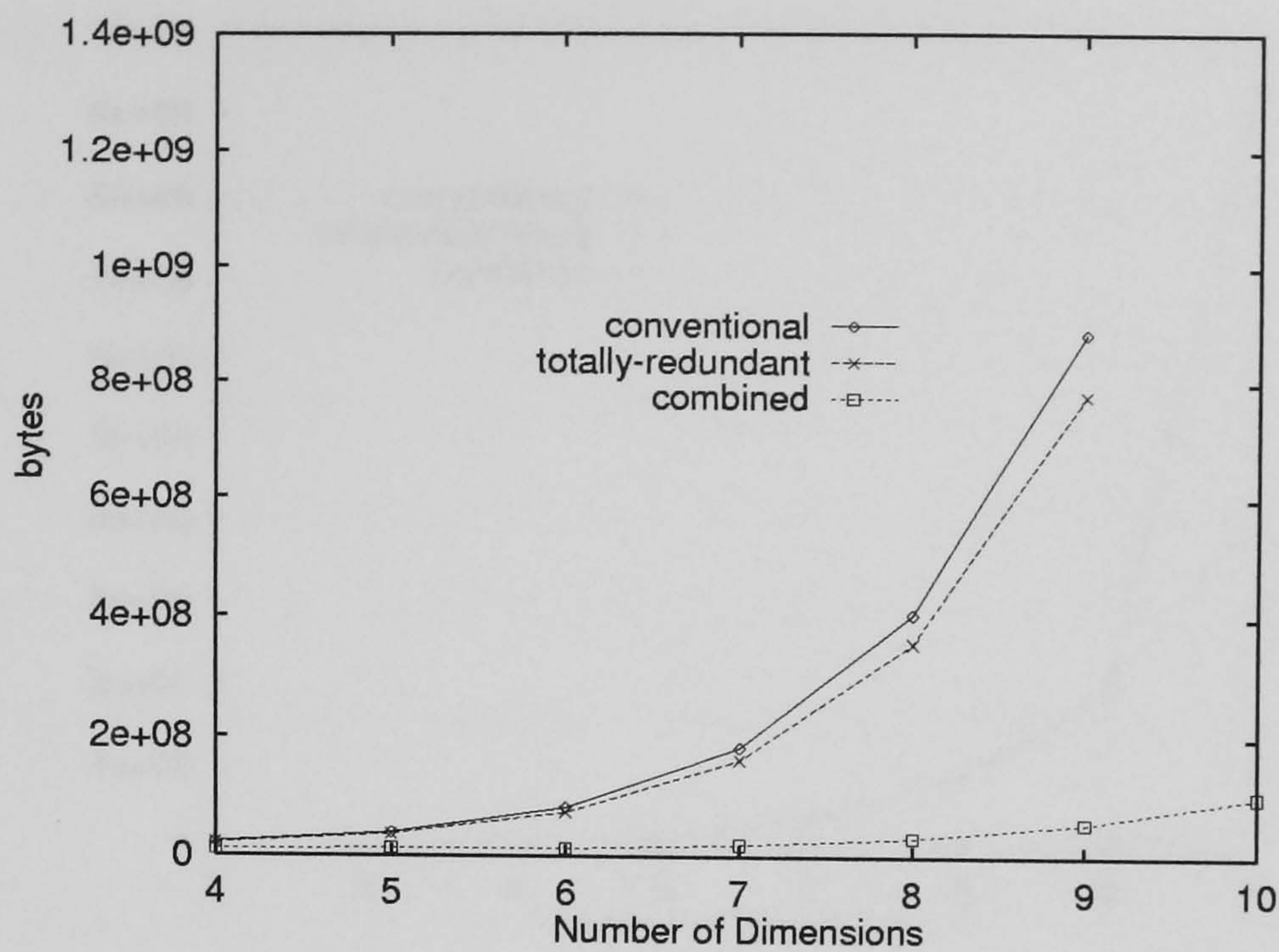

Figure C.11: Space savings of the L-R approach in the weather dataset

	Memory allocated (bytes)		
Dimensions	Conventional	Totally-Redundant	Combined
4	24,224,308	22,343,788	12,214,472
5	40,759,772	38,408,936	13,566,378
6	83,271,824	75,277,160	14,244,316
7	183,996,368	163,301,248	20,680,660
8	406,705,552	358,258,956	33,351,124
9	886,559,064	778,375,868	58,122,528
10	out of memory	out of memory	103,316,944

Table C.11: Space savings of the L-R approach in the weather dataset

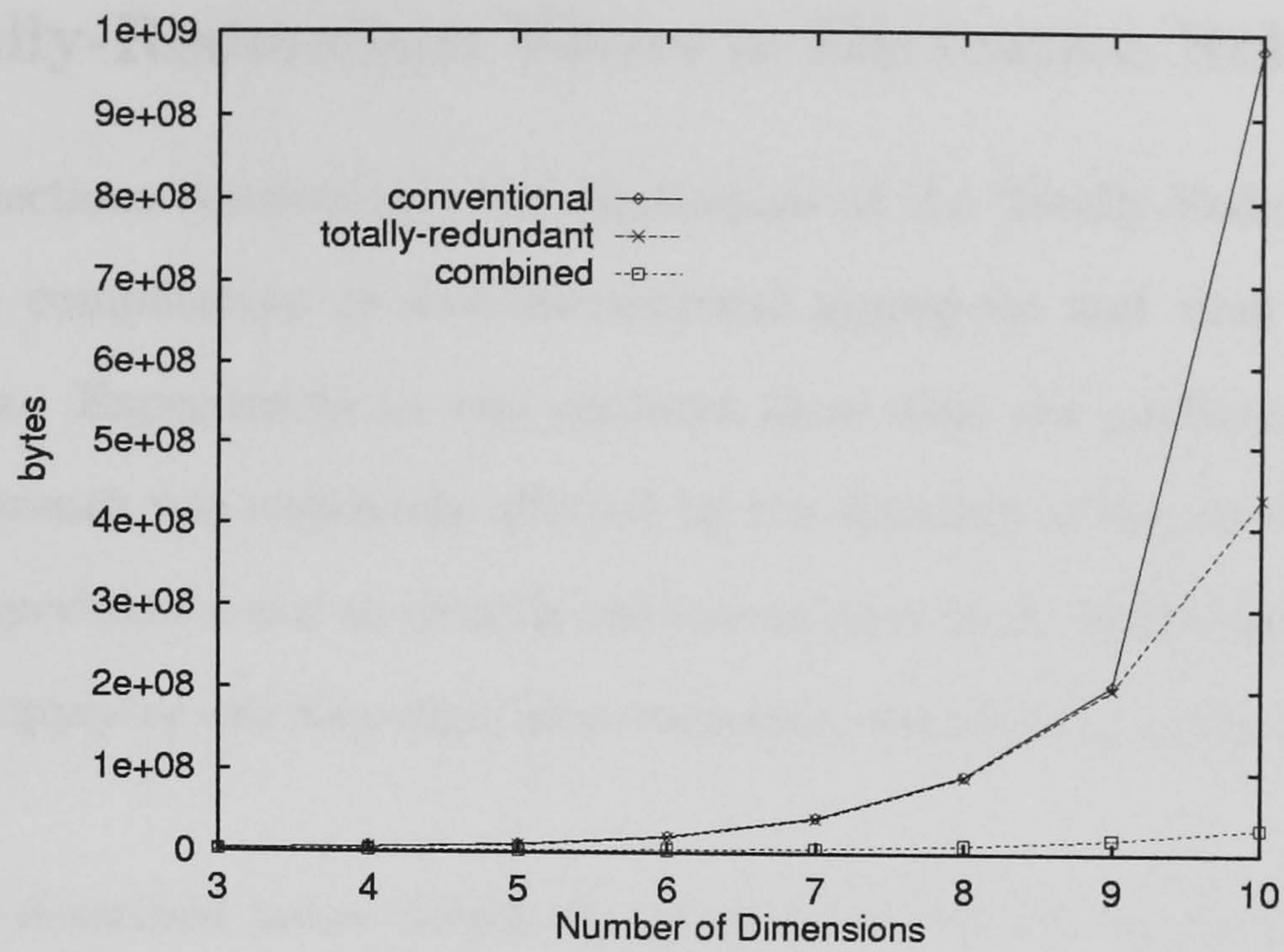

Figure C.12: Space savings of the L-R approach in the Adult dataset

	Memory allocated (bytes)		
Dimensions	conventional	Totally-Redundant	Combined
3	4,351,036	4,351,648	2,888,260
4	6,331,296	6,332,480	3,112,196
5	10,600,732	9,456,104	3,168,372
6	20,292,488	18,913,620	4,089,700
7	43,665,852	42,049,492	6,388,900
8	94,838,596	93,001,120	10,498,208
9	206,461,812	201,510,012	18,318,068
10	982,046,644	436,772,964	32,067,976

Table C.12: Space savings of the L-R approach in the Adult dataset

C.3 Totally-Redundant Views in Derivative Relations

The previous sections demonstrate the significance of the Totally-Redundant views approach for the computation of multidimensional aggregates and their storage as materialised views. Experiments in real datasets show that the performance of Totally-Redundant approach was negatively affected by the skewness of the datasets. The aim of this series of experiments was to identify the redundancy in derivative aggregate relations and show that applying the Key-algorithm recursively can lead to substantial increases in space savings.

The figures described below reveal the potential of the Totally-Redundant approach when the Key-algorithm is utilised recursively.

The synthetic datasets

Figure C.13 and Table C.13 compare the volume of the Totally-Redundant views when they are g-equivalent to the base relation with the volume of the Totally-Redundant views when they are g-equivalent to the derivatives. For the 600K dataset in seven dimensions, the results show a further redundancy of 23.5% compared to those found in the base relation.

Figure C.14 and Table C.14 illustrate the results of experiments using the TPC-D 60K dataset. This shows that redundancy is approximately 15% more than the redundancy found by the simple key algorithm when the derivative aggregates have been searched.

Figure C.15 and Table C.15 compare the same quantities for the 6K dataset. The redundancy found in this method is approximately 21% more than that found by using the input relation as a reference for the equivalence property.

For the hotel dataset, the redundancy in ten dimensions was 13.3% more than the simple (base relation) method. These results are illustrated in Figure C.16 and Table C.16.

The real datasets

The effect of the recursive Key-algorithm on the derivative relations using real datasets underpins the importance of the technique. Results for the weather dataset reveal that redundancy is 48.41% more than that found by the simple key algorithm. The simple Key-algorithm had identified only 10.26% of the data cube redundancy, compared to tests run in the derivative relations, in which the redundancy was 59.93%. Figure C.17 and Table C.17 show the results of six different data cube trials.

Figure C.18 and Table C.18 show results for the adult dataset. For the simple Key-algorithm, redundancy of Totally-Redundant views (based on the base relation) was 1.88%, in contrast to the redundancy identified in derivative relations of 65.28%.

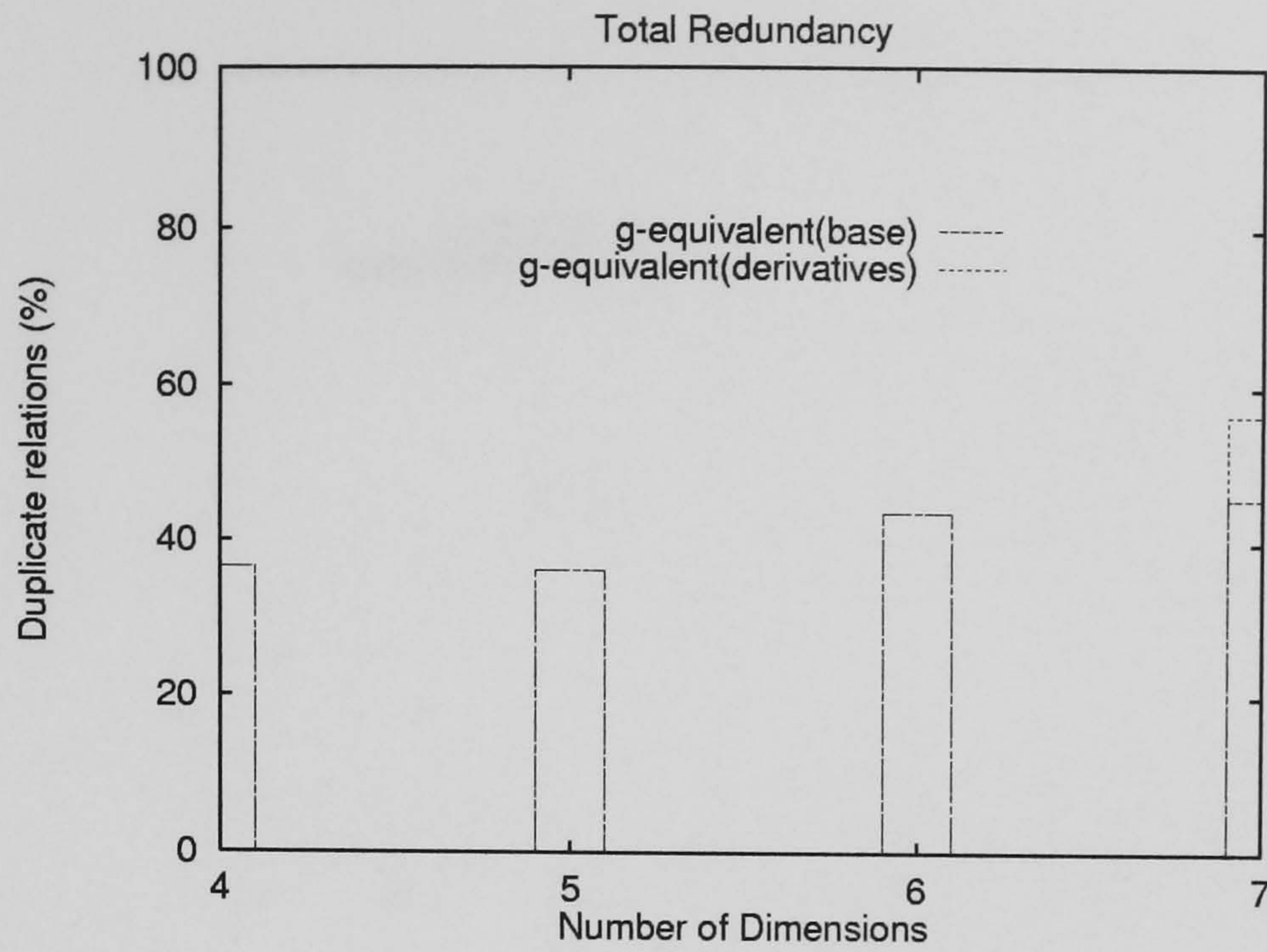

Figure C.13: The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (600K)

	Totally-Redundant views %	
Dimensions	Totally-Redundant(base relation)	Totally-Redundant(derivatives)
4	36.54	43.06
5	36.35	36.25
6	43.97	43.97
7	45.87	56.66

Table C.13: The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (600K)

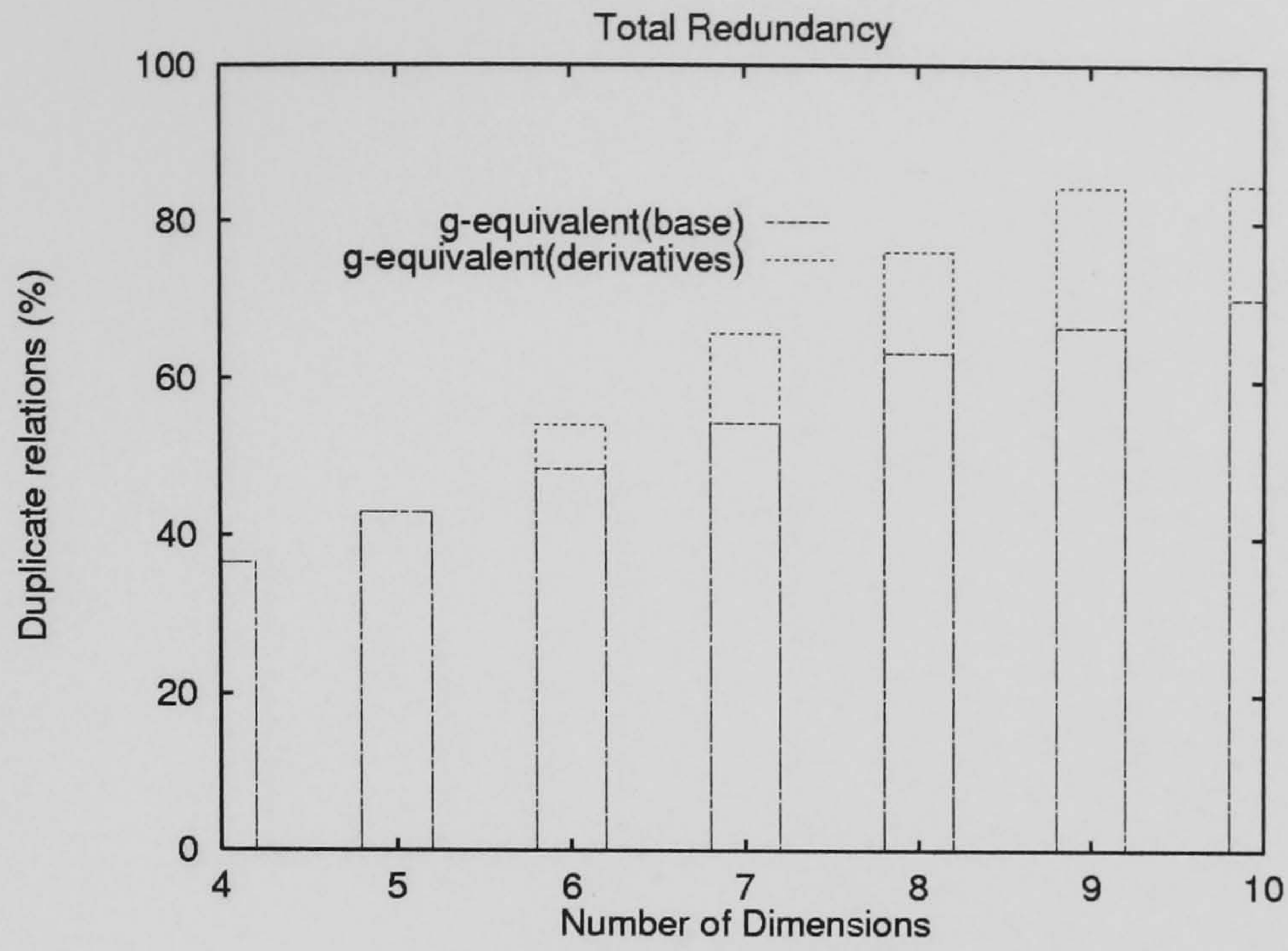

Figure C.14: The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (60K)

	Totally-Redundant views %	
Dimensions	Totally-Redundant(base relation)	Totally-Redundant(derivatives)
4	36.62	36.62
5	43.06	43.06
6	48.72	54.35
7	54.60	65.86
8	63.47	76.33
9	66.80	84.47
10	70.39	84.79

Table C.14: The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (60K)

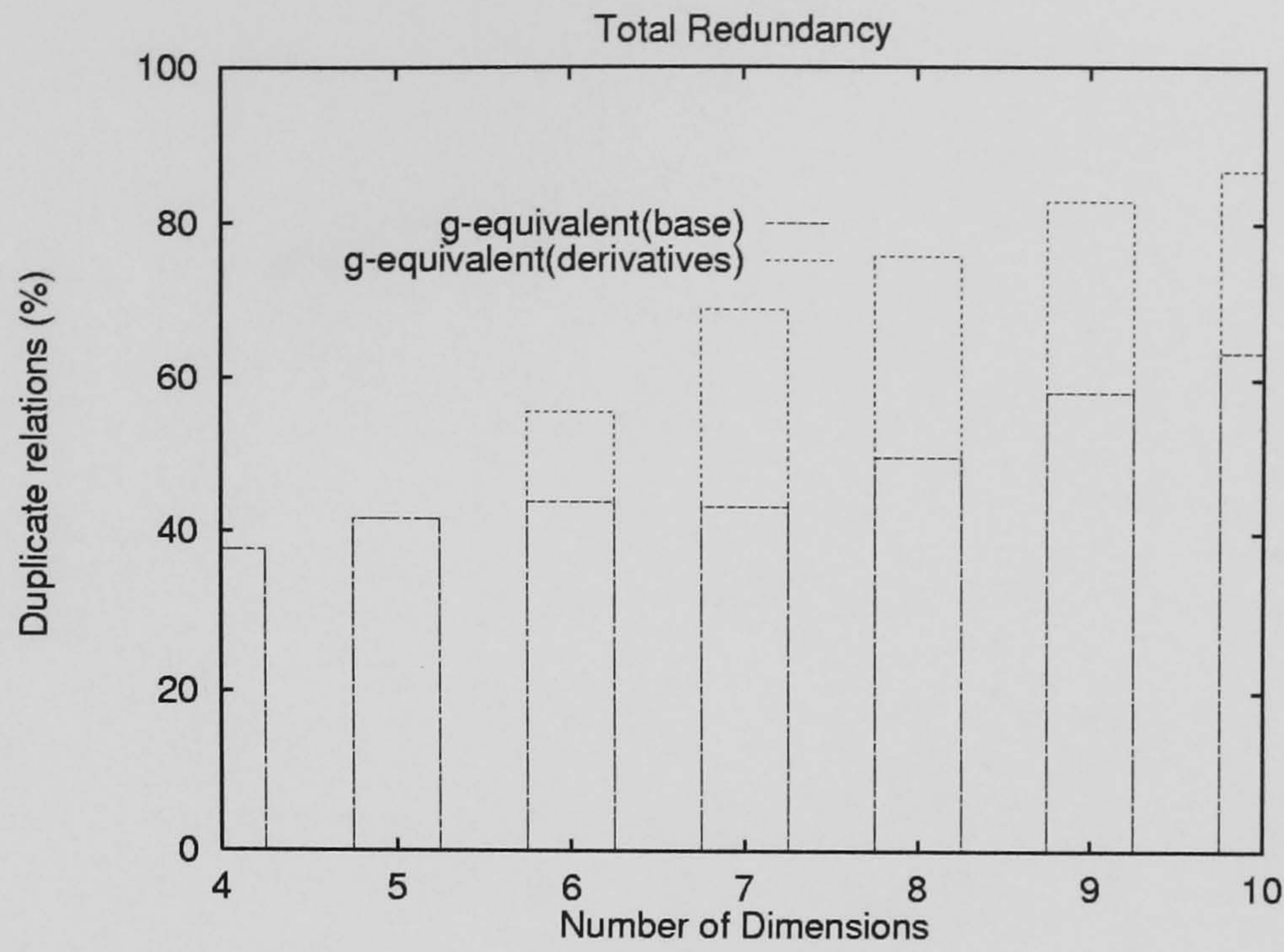

Figure C.15: The effect of Totally-Redundant views on space in the TPC-D Lineitem Table (6K)

	Totally-Redundant views %	
Dimensions	Totally-Redundant(base relation)	Totally-Redundant(derivatives)
4	37.75	37.75
5	41.65	41.65
6	43.94	55.59
7	43.33	68.91
8	49.76	75.84
9	58.34	82.90
10	68.18	89.70

Table C.15: The effect of Totally-Redundant views on space in TPC-D Lineitem Table (6K)

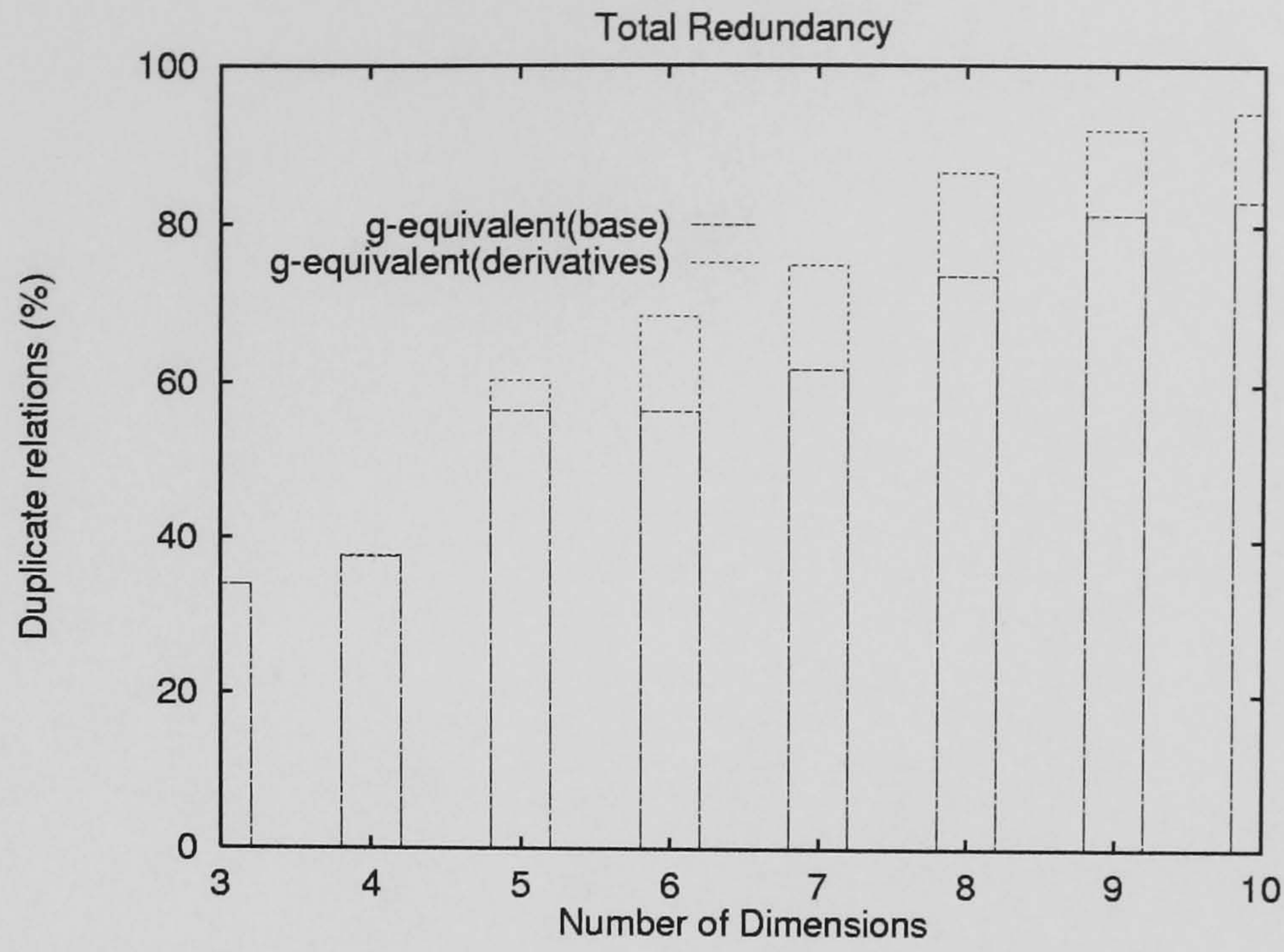

Figure C.16: The effect of Totally-Redundant views on space in the Hotel dataset

	Totally-Redundant views %	
Dimensions	Totally-Redundant(base relation)	Totally-Redundant(derivatives)
3	34.04	34.04
4	37.64	37.68
5	56.48	60.26
6	56.51	68.56
7	61.97	75.04
8	73.64	86.62
9	81.41	91.95
10	83.03	94.08

Table C.16: The effect of Totally-Redundant views on space in the Hotel dataset

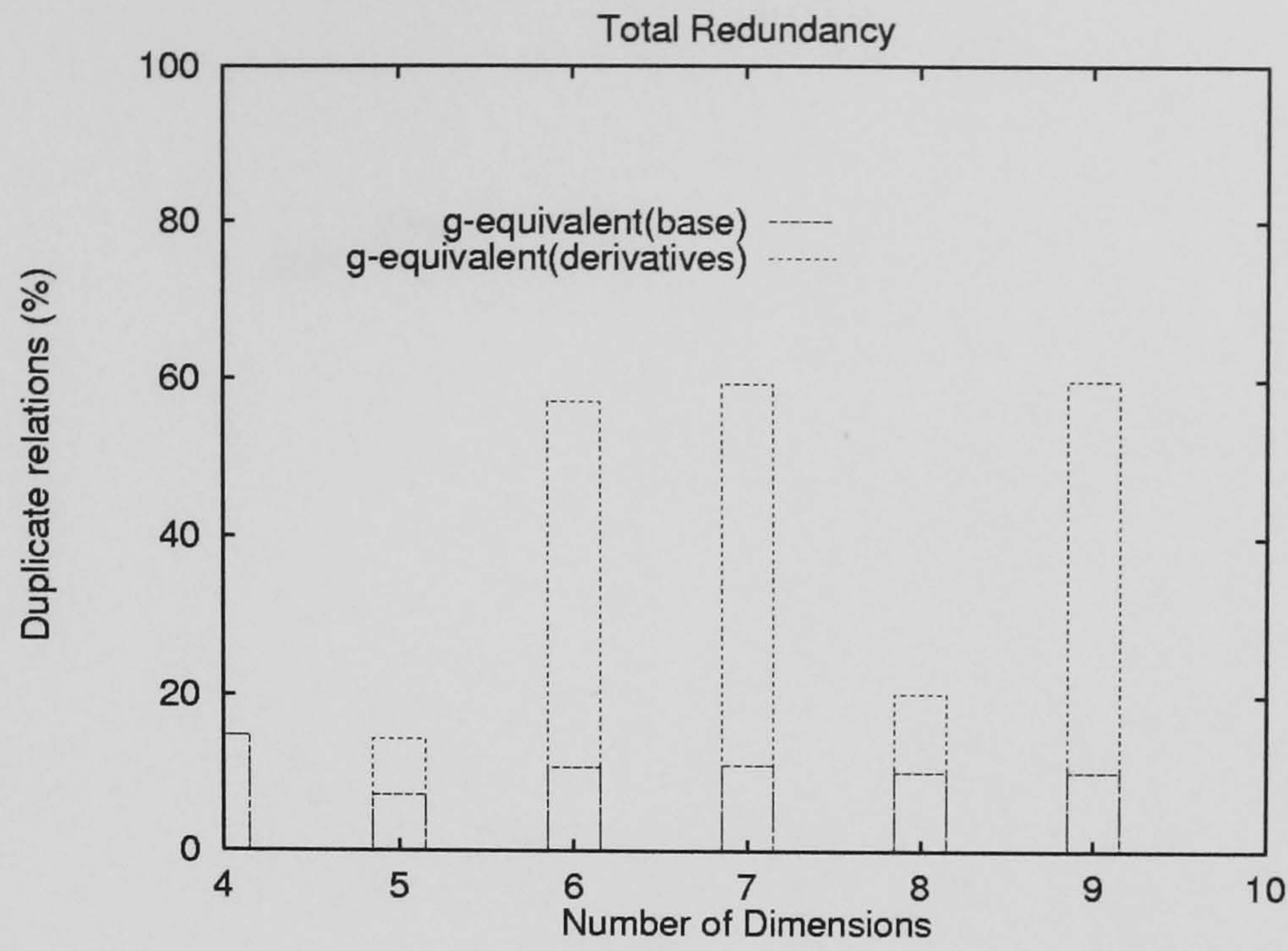

Figure C.17: The effect of Totally-Redundant views on space in the Weather dataset

	Totally-Redundant views %	
Dimensions	Totally-Redundant(base relation)	Totally-Redundant(derivatives)
4	14.91	14.91
5	7.199	14.39
6	10.79	57.19
7	11.13	59.54
8	10.19	20.39
9	10.26	59.93

Table C.17: The effect of Totally-Redundant views on space in the Weather dataset

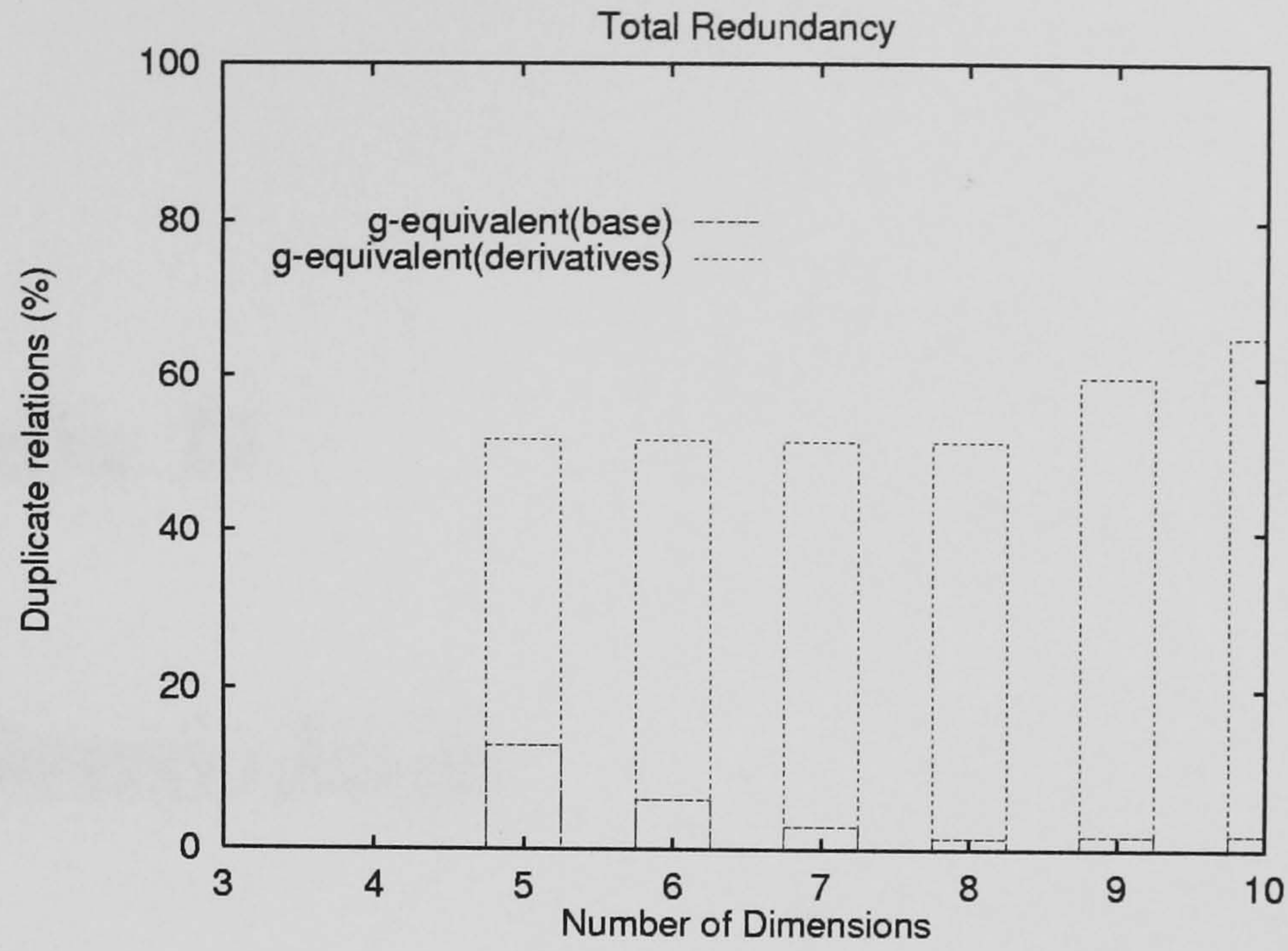

Figure C.18: The effect of Totally-Redundant views on space in the Adult dataset

	Totally-Redundant views %	
Dimensions	Totally-Redundant(base relation)	Totally-Redundant(derivatives)
4	none	none
5	12.95	51.91
6	6.28	51.80
7	2.94	51.66
8	1.42	51.72
9	1.73	60.06
10	1.88	65.28

Table C.18: The effect of Totally-Redundant views on space in the Adult dataset

Appendix D

The Semi-Join

Reconstruction of the aggregate requires a semi-join operation after aggregation, which is an expensive operation. Faster implementation of the semi-join operator can be achieved using the Bloomjoin [Bloom70] rather than a conventional join algorithm [ML86], [ME92]. This method is based on *Bloom filters* which is an array of bits *B[l..m]* with every bit initially set to zero. Given two relations, the approach for joins is described in the following stages. For the first relation, a hash function h, is applied to the join attributes. The hash value $h(s)$ points to a bit in the Bloom filter and this bit is set to 1. At the end of the process the array bits are either 1s or 0s. The array is then used to determine whether a given attribute value is present in the relation. For the second relation, the join attributes are hashed and if the hash value points to a bit set to 1 in the Bloom filter, the corresponding tuple is likely to have a match in the first relation. The Bloom-filter has been developed by [SL76] to screen out most accesses to a differential file for view maintenance purposes [Hans99]. [Babb79] and [MTD76] have also used bit-arrays for faster joins. However, applying this technique to extract the Difference representation from the aggregate relation would require the scanning of the two relations (parent and aggregate) for every Partially-Redundant relation. The aggregate algorithms described in section 3.6 achieve this extraction in a more efficient way.