ADI : A NoSQL system for bi-temporal databases

Azhar Ait Ouassarah

UNIVERSITÉ DE LYON

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

2016LYSEI046

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
préparée au sein de
l'INSA LYON

Ecole Doctorale ED512
Ecole doctoral d'informatique et mathématique de Lyon

Spécialité de doctorat : Informatique

Azhar AIT OUASSARAH

ADI: A NoSQL System for Bi-temporal Databases

Devant le jury composé de :

Roncancio, Claudia	Professeur	Grenoble INP-ENSIMAG	Présidente
Abdessalem, Talel	Professeur	Télécom ParisTech	Rapporteur
Vodislav, Dan	Professeur	Université Cergy-Pontoise	Rapporteur
Petit, Jean-Marc	Professeur	INSA-LYON	Dir de thèse
Scuturici, Vasile-Marian	MdC (HDR)	INSA-LYON	Co-dir de thèse
Averseng, Nicolas	VP R&D	Axway	Examinateur
Murisasco, Elisabeth	Professeur	Université du Sud Toulon-Var	Examinatrice
Revol, Romain	Ingénieur	Axway	Invité

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **http://www.edchimie-lyon.fr** Sec : Renée EL MELHEM Bat Blaise Pascal 3e etage secretariat@edchimie-lyon.fr Insa : R. GOURDON	M. Stéphane DANIELE Institut de Recherches sur la Catalyse et l'Environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 avenue Albert Einstein 69626 Villeurbanne cedex directeur@edchimie-lyon.fr
E.E.A.	**ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE** **http://edeea.ec-lyon.fr** Sec : M.C. HAVGOUDOUKIAN Ecole-Doctorale.eea@ec-lyon.fr	M. Gérard SCORLETTI Ecole Centrale de Lyon 36 avenue Guy de Collongue 69134 ECULLY Tél : 04.72.18 60.97 Fax : 04 78 43 37 17 Gerard.scorletti@ec-lyon.fr
E2M2	**EVOLUTION, ECOSYSTEME, MICROBIOLOGIE, MODELISATION** **http://e2m2.universite-lyon.fr** Sec : Safia AIT CHALAL Bat Darwin - UCB Lyon 1 04.72.43.28.91 Insa : H. CHARLES Safia.ait-chalal@univ-lyon1.fr	Mme Gudrun BORNETTE CNRS UMR 5023 LEHNA Université Claude Bernard Lyon 1 Bât Forel 43 bd du 11 novembre 1918 69622 VILLEURBANNE Cédex Tél : 06.07.53.89.13 e2m2@ univ-lyon1.fr
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTE** **http://www.ediss-lyon.fr** Sec : Safia AIT CHALAL Hôpital Louis Pradel - Bron 04 72 68 49 09 Insa : M. LAGARDE Safia.ait-chalal@univ-lyon1.fr	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 avenue Jean Capelle INSA de Lyon 696621 Villeurbanne Tél : 04.72.68.49.09 Fax :04 72 68 49 16 Emmanuelle.canet@univ-lyon1.fr
INFOMATHS	**INFORMATIQUE ET MATHEMATIQUES** **http://infomaths.univ-lyon1.fr** Sec :Renée EL MELHEM Bat Blaise Pascal 3e etage infomaths@univ-lyon1.fr	Mme Sylvie CALABRETTO LIRIS – INSA de Lyon Bat Blaise Pascal 7 avenue Jean Capelle 69622 VILLEURBANNE Cedex Tél : 04.72. 43. 80. 46 Fax 04 72 43 16 87 Sylvie.calabretto@insa-lyon.fr
Matériaux	**MATERIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec : M. LABOUNE PM : 71.70 –Fax : 87.12 Bat. Saint Exupéry Ed.materiaux@insa-lyon.fr	M. Jean-Yves BUFFIERE INSA de Lyon MATEIS Bâtiment Saint Exupéry 7 avenue Jean Capelle 69621 VILLEURBANNE Cedex Tél : 04.72.43 71.70 Fax 04 72 43 85 28 Ed.materiaux@insa-lyon.fr
MEGA	**MECANIQUE, ENERGETIQUE, GENIE CIVIL, ACOUSTIQUE** **http://mega.universite-lyon.fr** Sec : M. LABOUNE PM : 71.70 –Fax : 87.12 Bat. Saint Exupéry mega@insa-lyon.fr	M. Philippe BOISSE INSA de Lyon Laboratoire LAMCOS Bâtiment Jacquard 25 bis avenue Jean Capelle 69621 VILLEURBANNE Cedex Tél : 04.72 .43.71.70 Fax : 04 72 43 72 37 Philippe.boisse@insa-lyon.fr
ScSo	**ScSo*** **http://recherche.univ-lyon2.fr/scso/** Sec : Viviane POLSINELLI Brigitte DUBOIS Insa : J.Y. TOUSSAINT viviane.polsinelli@univ-lyon2.fr	Mme Isabelle VON BUELTZINGLOEWEN Université Lyon 2 86 rue Pasteur 69365 LYON Cedex 07 Tél : 04.78.77.23.86 Fax : 04.37.28.04.48

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Acknowledgements

This *PhD* thesis is a joint work between Axway company and LIRIS laboratory in a so called *CIFRE* funding.

Remerciements

Je souhaite tout d'abord exprimer ma profonde gratitude envers mon directeur de thèse Jean-Marc Petit et mon co-directeur de thèse Vasile-Marian Scuturici pour la qualité de leur encadrement, leurs conseils et leur disponibilité pendant toute la durée ma thèse.

Je tiens à remercier les professeurs Abdessalem et Vodislav d'avoir accepté d'être rapporteurs de ma thèse ainsi que les professeurs Murisasco et Roncancio d'avoir accepté de siéger au jury.

Cette thèse est le fruit d'une collaboration entre l'équipe Bases de Données (BD) du Laboratoire d'InfoRmatique en Images et Systèmes d'informatique (LIRIS UMR 5205 CNRS, INSA de Lyon - Université Lyon 1 - Université Lyon 2 - Ecole Centrale de Lyon) et l'entreprise Axway. La thèse a été financée par Axway et le ministère de l'enseignement supérieur et de la recherche à travers une convention CIFRE de l'ANRT. Je remercie Axway d'avoir soutenu cette thèse, et plus particulièrement Nicolas Averseng de m'avoir accordé sa confiance. Merci à Romain Revol et Xavier Fournet pour leur encadrement. Merci aussi à toute l'équipe ADI: Adrien, Arnaud, Guillaume, Mickaël, Sébastien, Sylvain et tous les autres. Cette expérience fût très enrichissante, et l'ambiance de travail très sympathique. Les petits-déjeuners du vendredi me manqueront.

Je remercie aussi les membres du LIRIS. Une pensée aux doctorants et docteurs que que j'ai côtoyés pendant ces trois années: Arnaud, Bilal, Diana, Lucile, Manel, Mazen, Sébastien, Tarek, Vincent... Bon courage à vous pour la suite de votre carrière.

Je remercie ma famille pour son soutien, en particulier mes parents, ma soeur et mon beau-frère. Mes pensées à deux de mes oncles qui nous ont quittés ces dernières années. Je remercie aussi les familles Bouchet, Hennequin et Zaouak qui m'ont été d'un grand soutien depuis que je suis en France. Enfin un grand merci à mes amis Aminah, Célia, Marion, Najib, Nicolas, Sami et Youssoufa.

Abstract

La complexité et la dynamique de l'environnement dans lequel évolue chaque entreprise requiert de la part de ses managers la capacité de prendre des décisions pertinentes dans un laps de temps très court afin de maintenir ou accroître son activité. Pour cela, l'analyse des données générées par l'activité de l'entreprise peut être une précieuse source d'information.

Ces dernières années, une nouvelle classe de systèmes d'aide à la décision est apparue pour relever ce défi: L'*Intelligence Opérationnelle* (IO) [1]. Son objectif est de permettre aux managers opérationels d'avoir une très bonne compréhension de la situation de l'entreprise, à travers l'analyse de l'activité passée et présente.
Dans ce contexte, les notions de temps et de traçabilité sont primordiales dans la compréhension de l'évolution de l'activité de l'entreprise à travers le temps.

Dans cette thèse, nous présentons *Axway Decision Insight* (ADI), une solution d'*IO* développée par l'éditeur de logiciels Axway. Le composant clé de cette solution est un SGBD orienté colonnes et bi-temporel développé en interne par l'entreprise pour répondre aux besoins spécifiques de l'IO. Ses capacités bi-temporelles lui permettent de gérer nativement aussi bien l'évolution des données dans la réalité modélisée (*temps de validité*) que l'évolution des donnés dans la base de données (*temps de transaction*).

Nous commencerons par présenter la solution ADI en nous focalisant sur deux éléments importants: 1) l'interface graphique qui permet la conception et l'utilisation d'ADI sans écrire la moindre ligne de code. 2) L'approche adoptée pour modéliser les données bi-temporelles.
Ensuite, nous présentons un benchmark pour ADI qui se base sur le benchmark pour bases de données bi-temporelles *TPC-BiH* [2].
Après cela, nous présentons deux optimisations pour ADI. La première redéfinit une requête bi-temporelle en: 1) un ensemble de requêtes continues pour calculer des aggrégations et dont les résultats sont matérialisés, et 2) une requête qui accède aux résultats maté rialisés. La deuxième optimisation ordonne l'exécution des opérateurs de jointure des plans de requêtes en utilisant un modèle coût basé sur des statistiques des données bi-temporelles.
Pour évaluer ces optimisations, nous avons effectué des expérimentations en utilisant notre benchmark, et qui ont démontré leurs intérêts.

Contents

List of Figures

List of Tables

Abbreviations

2TDB	**B**i-temporal **D**atabase
2TDBS	**B**i-temporal **D**atabase **S**ystem
ADI	**A**xway **D**ecision **I**nsight
BAM	**B**usiness **A**ctivity **M**onitoring
BI	**B**usiness **I**ntelligence
CEP	**C**omplex **E**vent **P**rocessing
DB	**D**ata**B**ase
DSMS	**D**ata **S**tream **M**anagement **S**ystem
DSS	**D**ecision **S**upport **S**ystem
ER	**E**ntity-**R**elationship
EER	**E**nhanced **E**ntity-**R**elationship
ETL	**E**extract **T**ransform **L**oad
NMS	**N**etwork **M**anagement **S**ystem
OI	**O**perational **I**ntelligence
OLAP	**O**n**L**ine **A**nalytical **P**rocessing
OLTP	**O**n**L**ine **T**ransaction **P**rocessing
RDBMS	**R**lational **D**ata **B**ase **M**anagement **S**ystem
UI	**U**ser **I**nterface

Chapter 1

Introduction

1.1 Context

Companies are operating in very dynamic and complex environments that require from their managers agility and ability to make proactive decisions, in order to maintain or improve their business. The available information generated by company's activities is exploding due to the increasing use of various technologies such as automated data collection, machine logs, emails, RFID, GPS, ... The "data deluge" represents a gold mine on which companies are sitting on. In consequence, there is a variety of decision support systems for decision-makers. The choice of the adapted one depends on several factors such as the nature of the activity or the range of the decision, e.g strategic or operational decisions.

Decision-makers may rely on Business Activity Monitoring (BAM) [3] to take operational decisions. BAM aims to provide real-time access to critical business performance indicators. Thus managers can have a deep insight of what is currently happening in their business and then take rapid and effective decisions. BAM gathers its information in real-time by analyzing data streams from multiple sources. BAM systems often relies on technologies such as Complex Event Processing (CEP) [4], Data Stream Management System (DSMS) or traditional Database Management System (DBMS). CEP aims to detect interesting patterns of events, e.g. if events A and B happen simultaneously, then C happens too. DSMS are intended to manage data streams and perform SQL-likree queries on them. Nevertheless the BAM is limited because they only focus on real time

information rather than using existing historical data and temporal data in their various forms. They do not give managers the necessary hindsight to compare the current organization activity behavior with its history. In consequence, it can be hard to detect threats and opportunities.

Exploiting historical data is traditionally covered by tools and systems from the Business Intelligence (BI) domain [5]. They enable managers to understand what happened in the past and help them to prevent the mistakes in the future by taking relevant long term and strategic decisions. A BI system accesses to historical and structured data sources in a batch-loaded approach and compute performance indicators that are usually stored in relational databases called data warehouses. This process is referred to as Extract, Transform and Load (ETL). It appears that BI is not intended to real-time use-cases since analyses can not be delivered in real-time. Besides, it does not offer enough agility to meet manager's needs to take operational decisions in very dynamic environments.

This information is not always well-exploited due to the lack of adapted Decision Support Systems (DSS). In this setting, a new class of systems has emerged in the decision support system galaxy called *Operational Intelligence* (OI) [1] to meet the challenge of capturing, storing, analyzing and visualizing efficiently historical and real-time data. This class of systems is intended to help manager to take operational decisions and is situated at the crossroads of

BI and BAM (Figure 1.1), aiming to answer questions that no one of them can easily answer. This new DSS class does not intend to compete with existing systems, but rather complete them. Thus OI enables organizations to:

- Handle both historical and real-time data within the same system which enables managers to understand what happened and what is happening in their organization.
- Benefit from both BAM's agility to adapt to business evolution and BI's analytical capabilities.
- *Early events detection* to take immediate actions to address threats and opportunities
- *Higher operational performance:* The improvement of the business decisions leads to operational cost optimization, higher revenues.

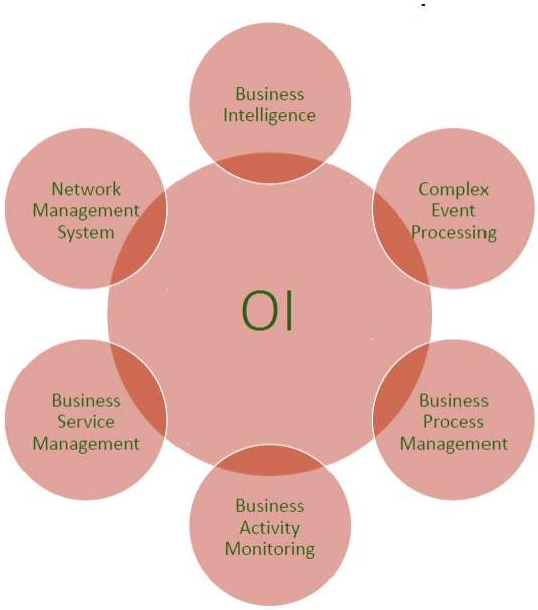


FIGURE 1.1: OI at the crossroads of disciplines

- *Better communication:* Managers knows exactly what has happened and what is happening in their business, and thus can better communicate and justify their actions.

Splunk[1], *Vitria*[2] Axway[3] are examples of companies that position theirself in OI.

All these functional features require to handle in an efficient way 2 temporal dimensions which are the history of data as well as the evolution of the DB modifications.

1.2 Bi-temporal Capabilities: Interest For OI

Databases are intended to store organization's knowledge concerning the real world. Most of these databases are considered as *static* [6] because they only store a snapshot of the world at a given time. As an example, let us consider the relation *customers* (Table 1.1a). Let us suppose that the customer *OneTwech*'s balance has been updated (Table 1.1b) from 400 to 300. From Table 1.1b, we do not know whether or not a change has occurred and when (if any).

[1] http://www.splunk.com/
[2] http://www.vitria.com/
[3] https://www.axway.com/en/enterprise-solutions/operational-intelligence

TABLE 1.1: Customer relation

(A) Before

customer_Name	balance
AxTech	100
OneTwech	400
Azeco	150

(B) After

customer_Name	balance
AxTech	100
OneTwech	300
Azeco	150

Classical databases systems are not intended to handle natively temporal data, but rather data at a certain time, usually the most recent one. In consequence, temporal DBs have been studied in the database community for more than three decades. Christian Jensen has identified around 2000 scientific productions over the 80^{th} and 90^{th} [7]. This includes among other things an encyclopedia [8], some books addressing the temporal databases topic [9–11], etc. These works have been referenced by several bibliographies such as [12].

Two main time aspects have been defined in the literature. They are *orthogonal*, which means that there is no clear relationship between them.

- The *valid time* "*vt*", also called *application time*, of a fact is the time when it is true in the modeled reality [13]. This time is intended to capture the history evolution of the modeled reality.
- The *transaction time* "*tt*", also called *system time*, of a database fact is the time when it is stored in the database [13]. It is intended to capture the history of database changes. It is consistent with the serialization order of transactions and is always provided by the DBMS. In consequence, the transaction times can not be later than the DB's current transaction time and can not be changed once a fact is timestamped in the database as for *vt*.

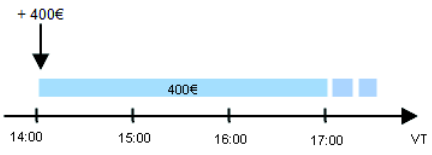
OI's capabilities that we have just detailed require to handle the history of business data. Such DMBS are known as a bi-temporal DBMS (2TDBMS), i.e a DBMS that natively supports both *valid time* and the *transaction time*.

1.2.1 Motivating Example For Bi-temporality

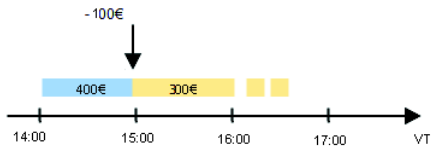
As a toy example, let us consider the relation *customers* (Table 1.1a). Let us suppose that the customer *Webtech*'s balance has been updated (Table 1.1b) from 200 to 300. From Table 1.1b, we do not know if a change has occured and when (if any).

Suppose that its balance is equal to 400€ at *14:00* the 1^{st} of January, 2016. The day will be omitted for clarity in the sequel. This information is stored in the DB as shown in Figure 1.2a where the balance evolution through an one dimension diagram.
At *15:00*, *OneTwech* places a new order costing 100€, and we assume that the balance value in the DB is updated instantaneously to 300€ (Figure 1.2b).
At *16:30*, *OneTwech* places a new order costing 500€ but it is rejected because its balance is too low (300€).
Let us say that *OneTwech* refunded its balance with 200€ at *16:00*. We suppose that the balance update process on the DB is instantaneous but for some reasons, it took 2 hours and the attribute is updated at *18:00* (Figure 1.2b).

FIGURE 1.2: OneTwech's balance evolution

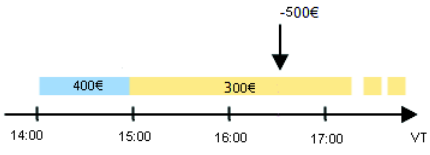


(A) at tt=vt=14:00: 400€

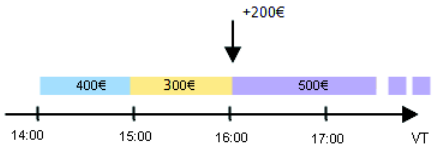


(B) at tt=vt=15:00: 100€

6



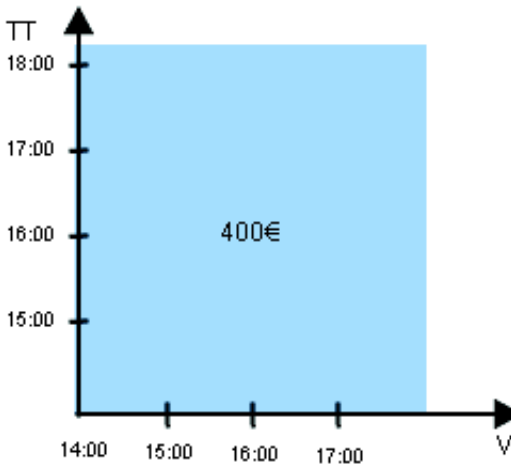
(C) new order rejected at 15:00



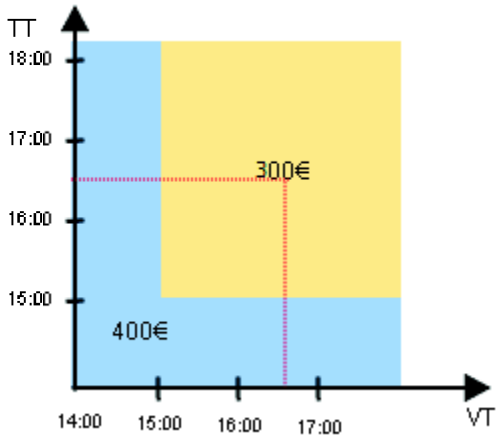
(D) At *18:00*, balance= 500€ from 16:00

When the manager analyzes its business activity sometimes later, he will notice that *OneTwech*'s order was rejected. From her point of view, this rejection is invalid because the balance account allows this order: Indeed 500€ are available at this moment (Figure 1.2d). However she can not know the reason because of the lack of information.

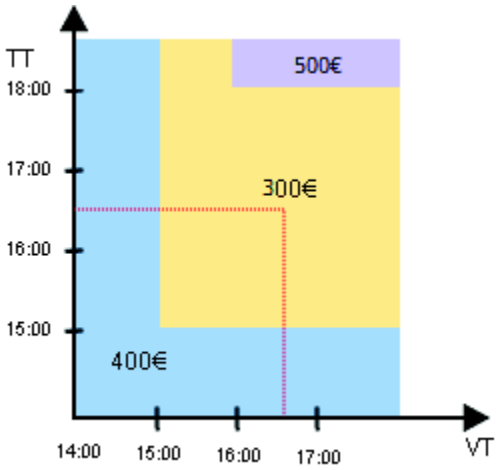
FIGURE 1.3: OneTwech's balance bi-temporal evolution



(A) tt=15 : 00



(B) tt=16 : 00



(C) tt=18 : 00

This example motivates the need of handling bi-temporal data to take rapid and relevant decisions.

Let us suppose now that we use a bi-temporal DB system and play the scenario again. Figures 1.3 summarizes the same information with a 2-dimensional diagrams (for tt and vt). Clearly, the manager can now access to all history of the balance and then understands why the order has been rejected at *16:30*.

1.2.2 Temporal Queries

The support of bi-temporal data, and more generally temporal data, enables new classes of queries [14]. We briefly discuss two of them:

- *Time travel*: One fixes both the valid time and the transaction to an instant. The query in Figure 1.3c is an example of such class which returns the balance's value at *vt=16:30* considering the DB at the *tt=16:30*.
- *Time range*: one fixes a temporal dimension to a particular instant, let us say the *transaction time*, while the other can either be fixed to an interval or vary over all the time domain. As an example let us consider Figure 1.4. The red line means that we want to get the history of *OneTwech*'s balance considering the state of DB at *tt=18:30*. The result is: {[14:00, 15:00[→ 400€, [15:00, 16:00[→ 200€, [16:00, ∞[→ 300€ }.

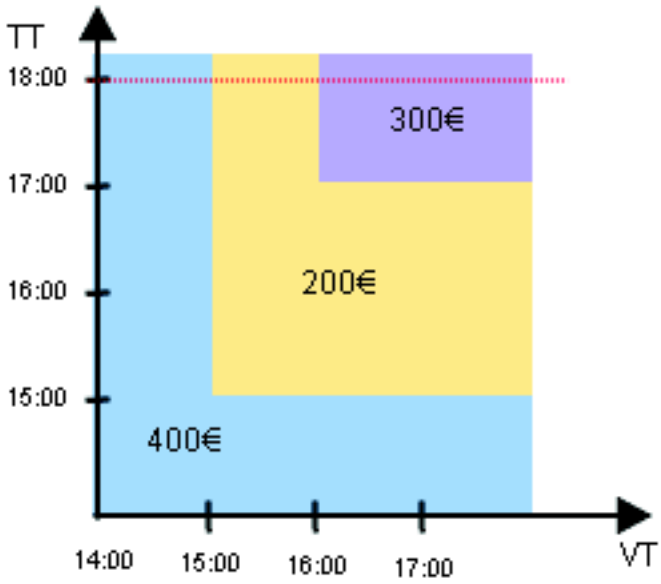


FIGURE 1.4: Graphical representation of a time range query

1.2.3 Temporal Aggregation

The temporal extension of DBMS raises the issue of defining the temporal aggregation computing contexts. As an example, let us consider Figure 1.5 that represents balance evolution of three customers according to the vt (tt is not considered to keep the

presentation simple). Answering the following query can be complex:

> What is the customer's average balance value during the interval [14:00, 17:00[?

Should we compute the average for each instant of that period or at some given instants? How should we handle data unavailability concerning *AxTech*?

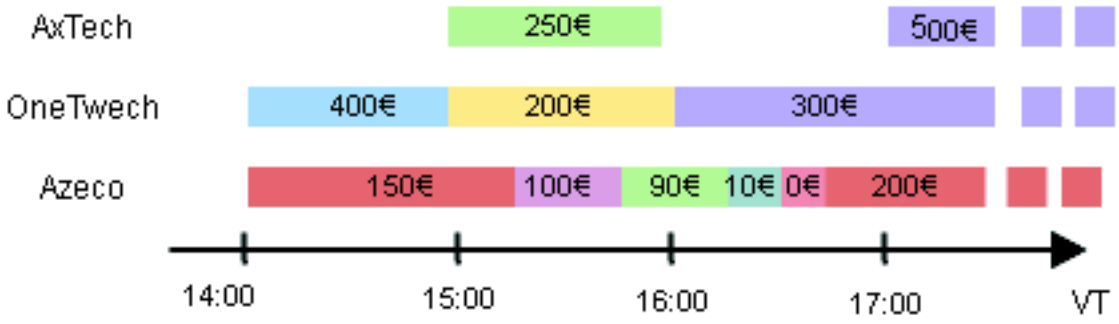


FIGURE 1.5: The temporal aggregation range

In order to overcome some of these issues, we introduce the concept of *Rhythm* which derives from the concept of *granularity* [15] and is similar to some concepts such as *time sequence attributes* [13], *time-series* [16] or *periodic attribute* [17]. It is a partition of the valid time domain into contiguous and equal-length time intervals. A rhythm is defined by a couple $(begin, duration)$ where *begin* is the reference time instant to be used for partitioning the valid time domain and *duration* is the length of each interval.
As an example, the rhythm (00:00,1 hour) corresponds to the following partition:

$$[00:00, 01:00[\cup [01:00, 02:00[\cup \ldots$$

If we use this rhythm for the previous query, then one possible approach would be to compute the aggregation at each hour during the interval [14:00, 17:00]. We assume that customers with unavailability data are not considered for the aggregation. In consequence the result of the query is: {14:00→ 275€, 15:00→ 200€, 16:00→ 195€, 17:00→ 333,33€ }

1.3 Axway Decision Insight

1.3.1 The Project Genesis

The genesis of Axway Decision Insight takes place in 2008 (Figure 1.6) when a French software editor Systar[4] launched a new project under the name of Tornado [18]. This latter was designed to be the company's next generation product and positions the company in the domain of OI. The project required more than 150.000 hours of R&D up to 2013 and was released in early 2013. In 2014, Systar was acquired by Axway, a top-5 French software editor[5], that claims to be a leader in data government flow. *Tornado* has been renamed to *Axway Decision Insight* (ADI) and became its spearhead in OI market.

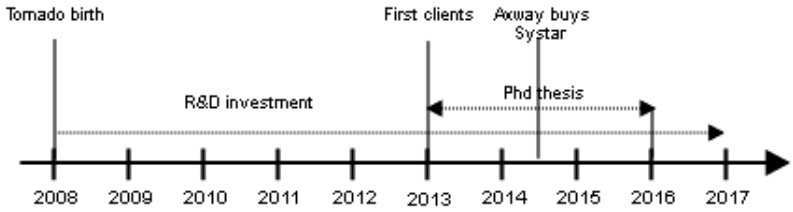


FIGURE 1.6: ADI project's chronology

1.3.2 Capabilities

ADI provides consolidated views of traditional real-time analyses with complete temporal historical analyses which help managers to have a deep understanding of what happened and what is happening in their organization. This leads to a higher operational performance allowing for example a revenue increase or an operational cost reduction.

ADI's 2TDBMS enables to support scenarii such as:

[4] http://www.systar.fr/
[5] http://www.truffle100.fr/2014/palmares.php

- Replaying past situations with exactly the same information as when they occurred live.
- Investigation for audit and traceability to provide:
  - In-depth analysis of a situation that occurred in the past, which could include answers to questions such as what happened, when, why and where specific actions are taken.
  - On demand simulations of the evolution of past situations;
  - Parallel comparison of the evolution in time of two situations, such as a present time process and the behaviour of the same process yesterday;
- Risk assessment evaluations, based on past of forecasted events and data.

1.3.3 Objectives

Since organization evolves in highly dynamic environments, decision-makers have to be agile. In consequence ADI has been designed to meet that requirement, by enabling:

- *Rapid time to value:* Applications should be implemented in significantly shorter time than traditional development cycles, typically under one month.
- *Low total cost of ownership:* ADI should achieve real-time monitoring with just one single platform running on commodity hardware and without using any other software.
- *Adaptability to changing business environment:* The ADI platform should easily be adapted in production applications as long as business evolves, thanks to a "code-free" approach. This means that a manager with limited technical skills in computer science can easily adapt its applications using a convenient graphical user interface (GUI) (Figure 1.7).

1.3.4 A Plug-and-Play Platform

ADI is a plug-and-play platform covering all the life-cycle of a monitoring application (Figure 1.8) that does not require any additional software. The main steps to implement an application are the following:

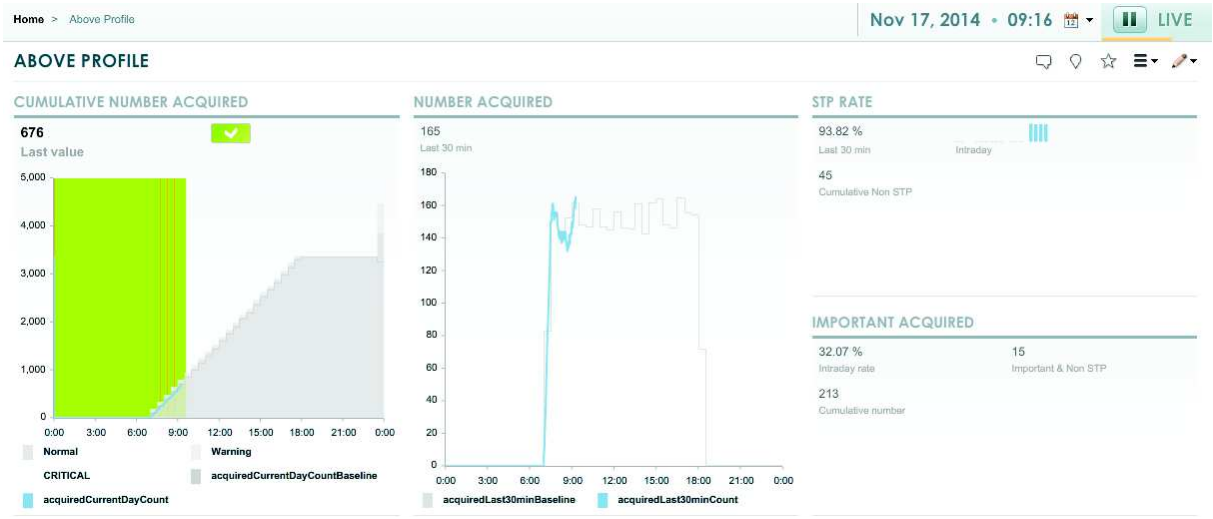


FIGURE 1.7: Example of an ADI's GUI

FIGURE 1.8: Application life cycle implementation

- *Designing a data model*: Decision Insight provides managers the possibility to model the data of their business using the Entity-relationship formalism. This formalism has the advantage of ignoring technical issues which corresponds to a no-technical manager's profil.

- *Implementing data integration routes*: Decision Insight provides several ready-to-use connectors to access to a wide range of data sources such as databases or web services.

- *Implementing analyses*: Managers may define their analyses thanks to a GUI tool on the top of the data model. Besides they can add or update an analysis while the application is in production.

- *Designing dashboards*: ADI provides managers with an interactive GUI that enables them to monitor their business by exploring both real-time and historical data (Figure 1.7). A dashboard is made up of one or more graphical elements (diagrams, charts, datagrids, ...) referred to as *pagelets* in the sequel. Each graphical element displays data returned by an underlying query. Managers can design themselves their dashboards using graphical tools. The dashboards can be updated while the application is running and see their modification instantly.

1.3.5 Architecture

The ADI Architecture (Figure 1.9) is based on a service oriented, event-driven implementation using *Java*. It is structured into three loosely coupled functional layers: *Absorption Layer*, *Logical Layer* and *User Interface Layer*.

The *Absorption Layer*: ADI uses unobtrusive, agentless technology to collect, process and analyse real-time data as well as historical data. It is based on the Apache Camel engine [6] that offers a wide range of possibilities to pull data from various types of sources.

User Interface Layer: The user interface layer allows any web browsers using Adobe Flash technology to display information from the logical layer. This layer is fully integrated to offer seamless and rich interaction with the analysts with visual data manipulation, navigation, as well as analyses based on its needs.

Logical Layer: Decision Insight is based on a proprietary DBMS, implemented by Axway, that is specifically designed to handle both real-time and historical data. This DBMS is bi-temporal and column-oriented. The bi-temporality means that it supports the *valid time* dimension to maintain the reality evolution and the *transaction time* dimension to maintain the database evolution. The column-oriented property means that data is stored according to columns, suitable for analytical use cases.

1.4 Thesis Challenges

When I started the thesis in December 2012, ADI was already implemented and started to be marketed. The product was internally implemented without any academic support, and this CIFRE contract was the first collaboration between Systar and a research laboratory. One of the thesis challenges was to compare and eventually align the concepts and vocabulary used in ADI with the state of art approaches at the international level. This includes the temporal data modeling approach as well as the query representation and processing. Beside there were also a concern about the positioning of ADI's DBMS compared with big software editors' products such as Microsoft, Oracle, SAP, etc.

Another challenge was related to ADI's DBMS performance issues. Supporting temporal features requires storing the whole history of data and not only its last version. Besides,

[6] http://camel.apache.org/

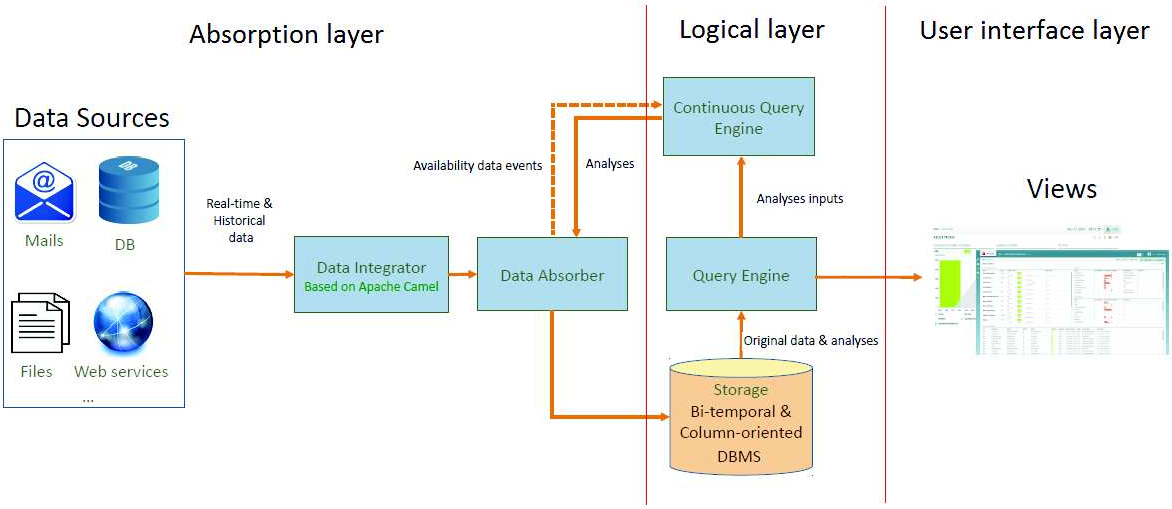


FIGURE 1.9: ADI's architecture

ADI provides users with an interactive GUI to monitor their business. It requires from the system to guarantee fast response time in order to avoid any GUI display lag which would make the platform unpractical, and thus reduces its interest. In this thesis, we mainly focus on the query engine part of ADI. Queries in ADI are specified through a declarative interface, implemented as an API instead of implementing a SQL-like language. This choice is mainly motivated by the fact that the DBMS is only intended to be used within ADI platform and for some particular, well-defined use cases.

Last but not least, the development of a piece of code within ADI is not an easy task. It has to deal with the code complexity induced by the platform complexity on which more than 20 engineers work every days. A simple modification of the code requires a wide variety of tests before to be pushed in the production platform.

1.5 Thesis Contribution

We can summarize the thesis's contribution in four points: *The alignment of ADI* with respect to the state of art, *the valorisation of ADI* in the international academic community, *the ADI benchmarking* and finally *query optimization*.

ADI's Alignment: This contribution consists in formalizing the existing product using the state of art in the column-oriented database field as well as in the temporal database field. Our studies concern the temporal extension of the entity-relationship model, the relational model and temporal functional dependencies [19].

ADI's Academic Valorisation: Despite the inherent difficulties to publish at the best international level in the database community, we published a demo at ICDE 2015 conference [18], a A* conference in the field. We also have presented a demo and a full paper at BDA 2015 conference[7].

Bi-temporal DB Benchmark: We have proposed a Bi-temporal DB benchmark adapted to OI use-cases to assess ADI performances. It is based on *TPC-BiH* [2], a bi-temporal extension of the well-known *TPC-H benchmark*.

Query Plan Optimization: The main academic contribution of this thesis is the optimization of bi-temporal queries. Up to now, ADI does not embed any cost-model based query optimizer. In consequence, we have proposed an optimizer that estimates the size of intermediate results generated by query plan's operator.

We also have explained how the ADI's bi-temporal query optimization module was working. It implements an optimization that consists in redefining complex bi-temporal queries into: 1) a set of continuous queries in charge of handling real time data streams (whose results are materialized) and 2) a query that accesses materialized results of the previous continuous queries. Thus, ADI can provide analysts with timely answers through a convenient GUI [18].

1.6 Document Organization

The remainder of this thesis is structured as follows:

[7] http://bda2015.univ-tln.fr/

- In Chapter 2, we introduce ADI. We first present its GUI that provides advanced features saving users from using writing a piece of code. We focus on two of them which are the ER editor used to design data models and the query editor used to design bi-temporal queries. Then we introduce modeling approaches used to handles bi-temporality and align it with the state of art. Then, we detail how data is physically handled. Finally we present some alternative modeling approaches, including conceptual and relational ones, that exist in the state of art.

- In Chapter 3, an adapted version of the TPC-BiH benchmark to OI use case is presented. We first address existing proposals of temporal DB benchmarks. Then we detail how we adapt the benchmark to meet ADI's requirements.

- In Chapter 4, we present a query optimization that limits GUI display lag by continuously precomputing queries' aggregation operations as data is collected. The experiments are conducted using the adapted TPC-BiH.

- In Chapter 5, we present a query plan cost-based optimizer for ADI. We first describe ADI's architecture to process data and queries. Then, we detail the statistics about data we collect and formulas to estimate the result size of query plan operators. Finally we detail the results of the experiments.

Chapter 2

Axway Decision Insight

2.1 Context

Axway Decision Insight (ADI) is an OI solution that provides consolidated views of traditional real-time analyses with complete temporal historical analyses. This product helps managers to have a deep understanding of what happened and what is happening in their organization and take relevant operational decisions. ADI's main innovation is a bi-temporal DBMS that has been specifically designed to meet OI requirements.

In this chapter, we introduce ADI's GUI that helps users to easily design their application. This includes the ER formalism to model applications as well as how to design bi-temporal queries. Then we present the logical model behind the ADI and how it physically stores temporal data. Finally, we present some alternative approaches to model temporal data at the conceptual level as well as at the logical one.

2.2 ADI: A Code-Free Platform

2.2.1 Conceptual Modeling

ADI provides managers with an ER graphical editor with bi-temporal capabilities to implement their application (Figure 2.1). The choice of both a graphical editor and a conceptual model is justified by the fact that managers have usually limited technical skills need to quickly implement applications.

The approach adopted to model temporal aspects is based on the usage of the classical ER model. It is motivated by the fact that all implemented databases on ADI are fully bi-temporal, and in consequence it is unnecessary to overload diagrams with additional annotations or constructs. It also avoids users from mastering additional constructs that are not contained in the original ER model.

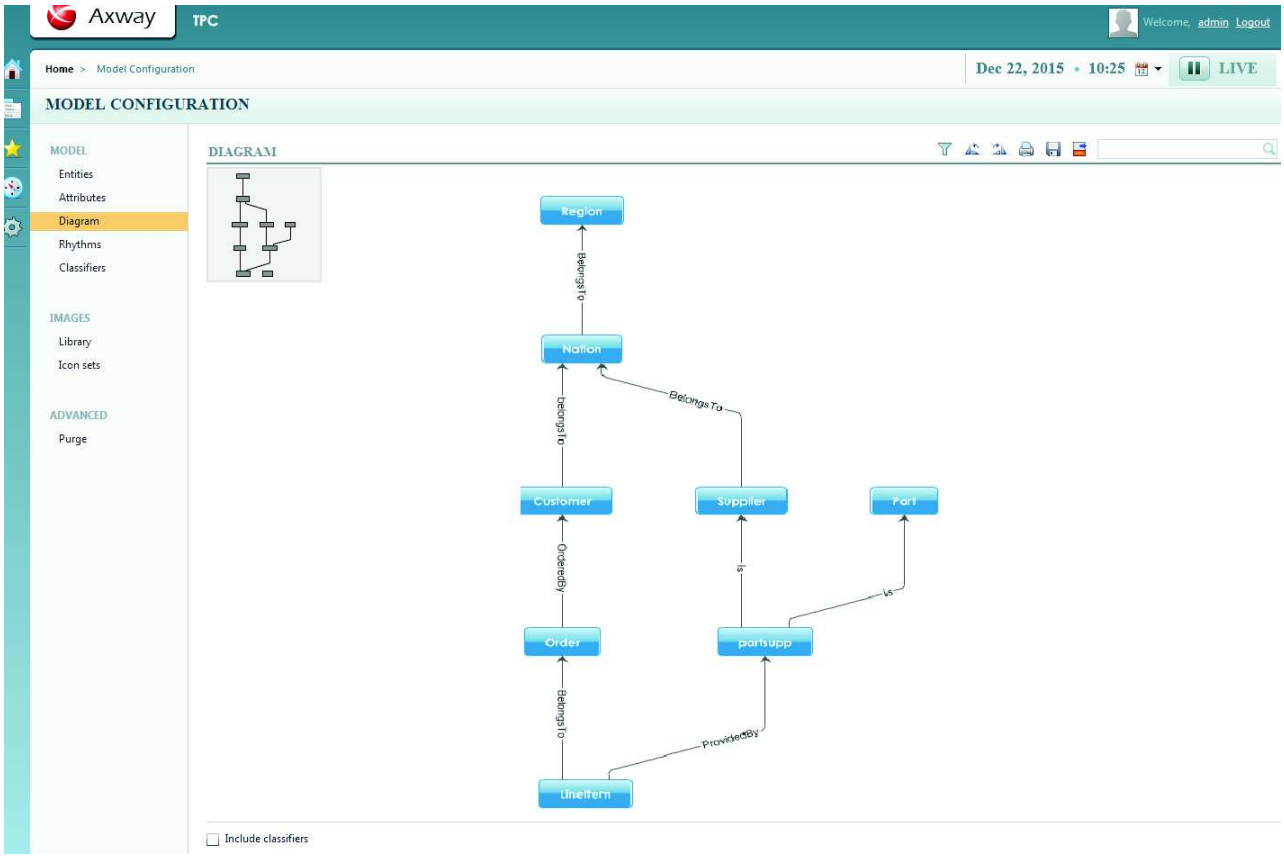


FIGURE 2.1: ADI's graphical user interface (GUI) to implement an ER diagram

2.2.2 Query Design in Decision Insight

Implementing queries using a SQL-based language can be a very difficult task, particularly for business managers with limited technical skills. ADI provides an advanced graphical interface for rapid design of the complex queries related to OI. As an example of such feature, let us consider the following query:

> "What is the total revenue achieved by the company every day during the current month (May 2015 in the example) considering the database at the most recent state?"

This query implementation is done in two steps. In the first step, the user implements the temporal aggregation through the GUI shown in Figure 2.2:

- (A) indicates which rhythm (temporal granularity used to compute the aggregation) we want to link to the attribute. In our case we choose an one-day rhythm as we want to know the company's total revenue per day.
- (B) indicates the aggregation operation used to generate the analysis.
- (C) indicates the time-range to consider for the aggregation. In our example we fix at the last day.
- (D) represents data sources used to compute the analysis, which is the attribute "extendedPrice" of the LineItem.

In the second step, the user chooses the form and the content that will be displayed on the "pagelet". Figure 2.4 is an example of ADI's GUI to define the pagelet to display the query result:

- (A) indicates graphical element type that the manager wants to display, namely a historical curve.
- (B) indicates the time range of information to display on the pagelet. According to the query, we choose to display the whole current month.
- (C) indicates the information to be displayed. Based on the provided information.

ADI creates a pagelet and an underlying *on-demand query* in charge of updating the pagelet content (Figure 2.3).

2.3 ADI's Physical Storage

ADI's DBMS is a bi-temporal and *column-oriented DBMS* [20] which has the particularity of being *attribute timestamped oriented* and *state-based*. In this section, we detail the reasons for choosing a column-oriented approach and we also define the *attribute-timestamping* and *state-based* approaches.

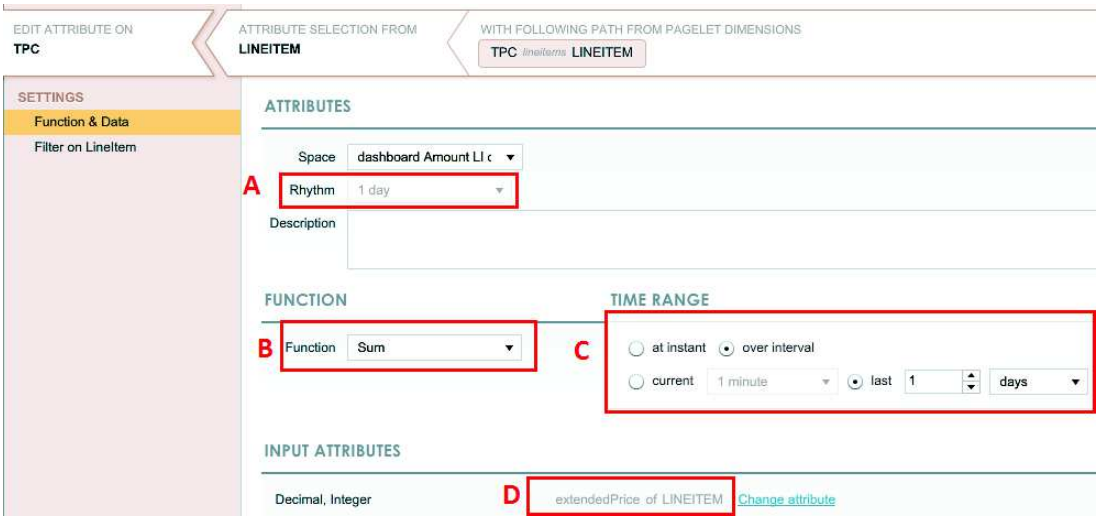


FIGURE 2.2: ADI's GUI to implement a temporal aggregation

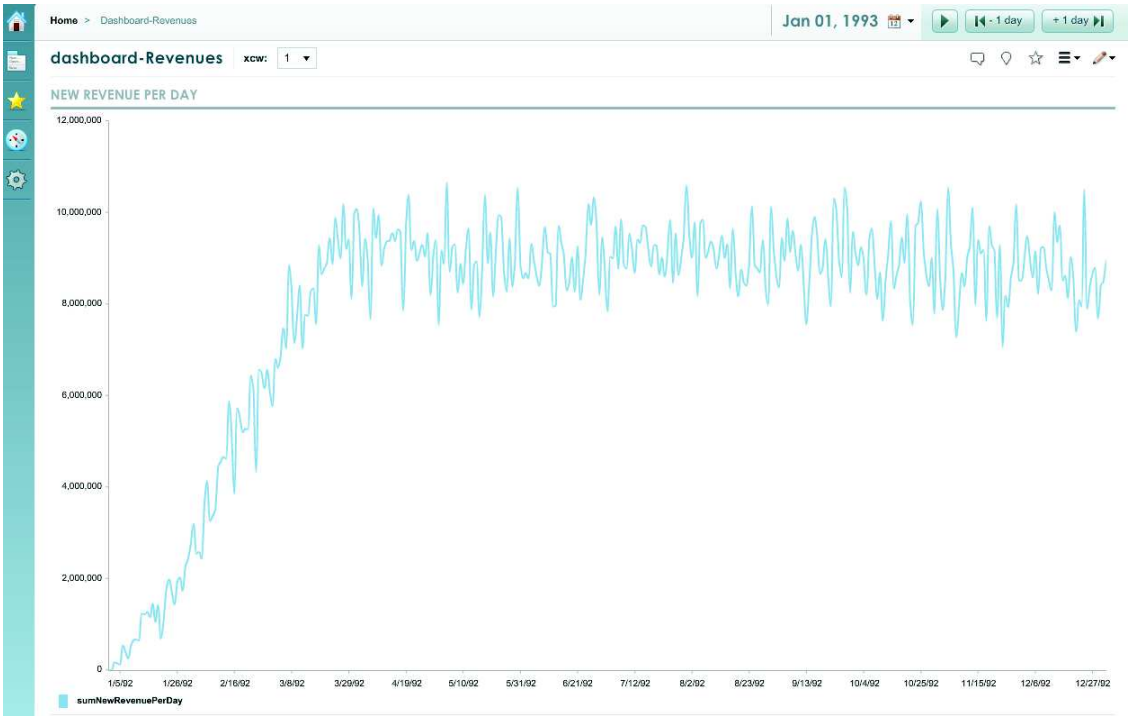


FIGURE 2.3: Snapshot of a pagelet displaying the daily evolution of revenues

2.3.1 Motivations for a column-oriented DBMS

Column-oriented DBMS appeared in 90*s* as a response of an increasing need for storing and analyzing very large volumes of data. The main difference between the classical (*row-oriented*) DBMS and the *column-oriented* ones is how data is physically stored in the system. A row-oriented DBMS stores data as rows of tuples. In a column-oriented,

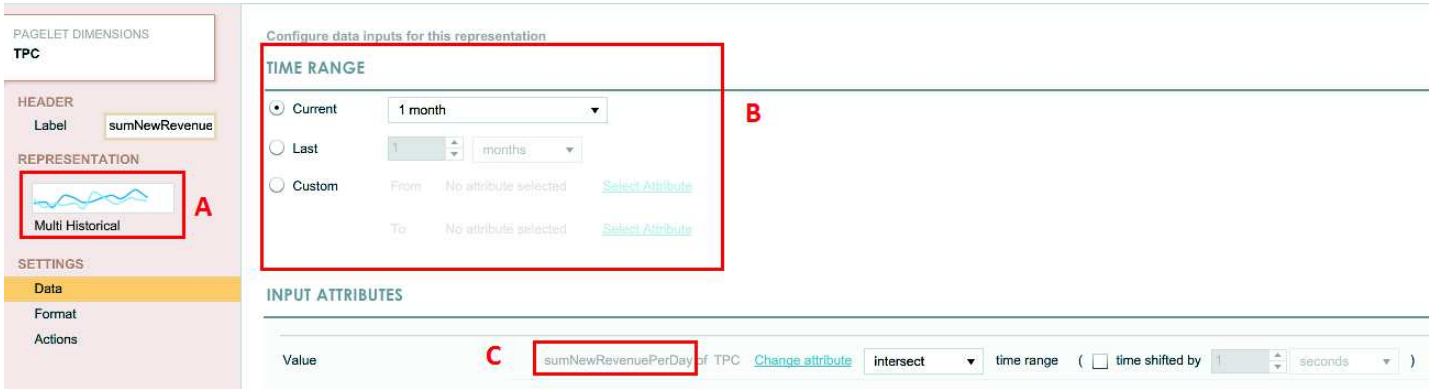


FIGURE 2.4: ADI's GUI to implement a pagelet

data is stored by columns. We outline below three main reasons motivating the column-oriented choice:

- The *column-oriented* databases are intended to perform analytical queries that analyze data and give an insight into the business activity, e.g the number of orders in pending status. The column-oriented database systems outperform row-oriented database systems on analytical workloads such as those found in business intelligence and decision support applications [21].
- The frequent evolution of OI applications, e.g GUI evolution, induces adding and removing attributes dynamically. The row-oriented approach is not suitable in this case because addition or deletion of an attribute affects the whole relation, with performance impacts on the modified relation. However the column-oriented approach does not suffer from this issue since each attribute is stored independently of each other.
- The will to handle in an independent way the temporal evolution of attributes. In the row-oriented approach, the update of an attribute value requires adding a new tuple with the new value. This behavior causes both a storage overhead and an increase in query execution time due to data duplication.

2.3.2 Attribute-Timestamping Approach

ADI's DBMS adopts an *attribute-timestamping* approach, also known as the *non first normal form* model. It consists in extending attributes from simple values to complex values that incorporate temporal dimensions. Among the models of this approach, we

can quote Gadia's model [22], McKenzie's model [23, 24] and Tansel's one [25]. To illustrate this approach, let us consider Gadia's model. A bi-temporal relation scheme R is defined as follows:

$$R = (\{([T_s, T_e] \times [V_s, V_e]A_1)\}, ...\{([T_s, T_e] \times [V_s, V_e]A_n)\} \tag{2.1}$$

Each tuple is composed of n sets. Each set element e is a triple of a transaction time interval [T_s, T_e], a valid time interval [V_s, V_e] and an attribute value $e.A_i$. An example of such relation is depicted in (Figure 2.5). The attribute-value timestamping approach

Customer	Balance	Nation
[1,∞]x[1,∞] Axtech	[1, 4]x[1,∞] 10 [5,∞]x[1,2] 10 [5,∞]x[3,∞] 30	[1,5]x[1,∞] Spain [6,∞]x[1,2] Spain [6,∞]x[3,∞] France
[2,∞]x[1,6] WebTech	[4,2] x[1,6] 9 [3,∞] x[1,1] 9 [3,∞] x[2,6] 12	[2,∞]x[1,6] Germany
[3,4]x[1,6] Azco	[3,4]x[1,6] 9	[3,4]x[1,6] France
[8,∞]x[4,∞] Jean Martin	[8,∞]x[4,∞] 9	[8,∞]x[4,∞] France

FIGURE 2.5: Example of Gadia's model relation

avoids any data redundancy because each attribute is handled separately. However this approach induces a storage overhead because each attribute is overloaded with additional temporal attributes. Besides, the models implemented using this approach may not be adaptable to the existing relational structures or to query evaluation techniques that suits for atomic values [26]. The choice of using either events or states to represent data in temporal relations depends on its expected use in applications. In the conventional relations, i.e relations with non temporal support, the reality is modeled as a single state that represents the most recent data. It is then natural to consider the states as the adapted approach to represent temporal relations. Yet, the event-based approach can be adapted to some particular use cases. One of them is when the database is append-only, i.e that once data is inserted in the DB, it can not be changed. In this case, data can be stored as events. In consequence it might be interesting to use the event-based approach to represent data as it is stored.

2.3.3 State-based Modeling Appproach

The support of time-varying data leads to a representation dilemma due to the existence of two opposite concepts: *states* and *events* [27, 28]. A *state* is something that lasts over time. It corresponds for example to a fact that is true during a time interval, but is not true before or after. An *event* however is instantaneous [13], i.e that occurs at a certain instant and does not last. A state is delimited by events. It starts when an event occurs and makes a fact true, and it ends when another event makes it false. In consequence a *state* can be represented by its delimiting events Figure 2.6.

Snodgrass' tuple timestamped representation scheme [29] is an example of state-based

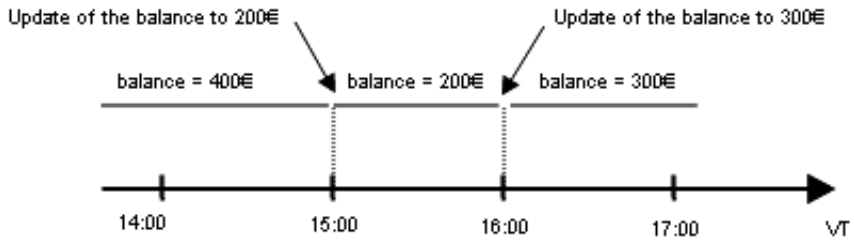


FIGURE 2.6: The duality of states and events

relational model approach. The relation schema R is defined as follows:

$$R = (A_1, A_2, ..., A_n, T_s, T_e, V_s, V_e)$$

A_1, A_2, ..., A_n are non temporal attributes. T_s and T_e are the bounds of the transaction time interval while V_s and V_e are the bounds of the valid time interval. Figure 2.7 is an example of a Snodgrass's proposal which adopts a *state-based* modeling approach.

The first tuple indicates that AxtTech's balance is 10 starting with the timestamp 1 and is still true, and this information is recorded at 1 and logically deleted at 5. The second tuple indicates that the balance is 10 from 1 to 3, fact known starting with the timestamp 5 and is still current in the DB.

In the conventional relations, i.e relations with non temporal support, the reality is modeled as a single state that represents the most recent data. It is then natural to consider the states as the adapted approach to represent temporal relations. This approach is the

Customer	Balance	Nation	V_s	V_e	T_s	T_e
AxTech	10	Spain	1	∞	1	5
AxTech	10	Spain	1	3	5	∞
AxTech	30	Spain	3	∞	5	6
AxTech	30	France	3	∞	6	∞
WebTech	9	Germany	1	6	2	3
WebTech	9	Germany	1	6	2	3
WebTech	9	Germany	1	2	3	∞
WebTech	12	Germany	3	6	3	∞
Azeco	9	France	1	6	3	4
Jean Martin	9	France	4	∞	8	∞

FIGURE 2.7: Example of Snodgrass's modeling approach

most common approach and is also adopted by Oracle's historical versionning module: *Workspace Manager*[1].

2.3.4 Translating an ER Diagram to a Column-store

In ADI, a column can either be an *Entity-column*, *attribute-column* or *relationship-column*.

- *Entity-column*: This column stores the instances of an entity type of the ER model. A record of this column is in the form of $< key, interval_{vt}, instant_{tt} >$. *key* is a surrogate attribute that uniquely identifies an instance of an entity type. The attribute $interval_{vt}$ is the lifetime of the instance and $instant_{tt}$ is the time when the record is inserted into the DB.

- *Attribute-column*: It stores the values of an attribute of the ER model. A record of this column is in the form of $< key_1, key_2, ..., key_n, interval_{vt}, instant_{tt}, value >$. The attributes key_i with $i \in \{1, ..., n\}$ are the surrogate attributes that refer to *entity-columns* to which the attribute belongs. The attribute $interval_{vt}$ is the time during which the record is valid, $instant_{tt}$ is the time when the record was inserted in the DB. An attribute is *monodimensional* if $n = 1$, and is *multidimensional* if $n > 1$.

- *Relationship-column*: It stores the value of a relationship of the ER model. A record of this column is in the form of $< key_1, key_2, interval_{vt}, instant_{tt} >$. key_1

[1] http://www.oracle.com/technetwork/documentation/index-087067.html

and key_2 refer to the entity-types involved in the relationship while $interval_{vt}$ and $instant_{tt}$ are respectively the time during which the record is valid and the instant the record is inserted in the DB.

2.3.5 Column Access API

The column store provides a catalog to get access to the desired column. For each column, there is an API to manipulate them. It mainly consists of the four following methods:

- *create(*$surID_1$*, ... *$surID_n$*, *$instant_{tt}$*, *$interval_{vt}$*, value)*: It inserts a new record to the column. "$surID_i$" with $i \in 1..n$ is a surrogate attribute [30] that uniquely identifies real word entities. The number of surrogate attributes depends on the column type, e.g an entity-column will only contain one surrogate attribute while a relationship column will contain at least two of them. "$instant_{tt}$" is the tt instant when data hold by the record was acquired by ADI and "$interval_{vt}$" is the time during which it is valid. The attribute "*value*" concerns only attribute-columns.

- *get(*$instant_{vt}$*, surID, *$instant_{tt}$*)*: It returns at most one record r, such that, its surID matches the method parameter's *surID*, $r.interval_{vt}$ intersects $instant_{vt}$ and it is the most recent with $r.instant_{tt} < instant_{tt}$. As an example let consider (Figure 2.8) which represents the stored entries of an attribute column for a given surrogate id, e.g $surID = 1$. A right arrow means that $vt_e = \infty$. The execution of $get(8, 7, 1)$ returns the result $\{(1, [1, \infty[, 1)\}$

- *scan(*$interval_{vt}$*, *$interval_{surID}$*, *$instant_{tt}$*)*: it returns all records, such that each record r, $r.surID \in \{surID_1, ..., surID_n\}$, $r.interval_{vt} \cap interval_{vt} \neq \emptyset\}$, $r.instant_{tt} \leq instant_{tt}$. As an example, let us consider the example in Figure 2.8 that represents an entity-column's records. A segment means that the vt is a closed interval while a right arrow means that it is a left-bounded one. The number on segments and right arrows represent the record's *surID*. If we consider the call "$scan([3, 9], [4, 8], 8)$", then the predicate "$vt = [3, 9]$" returns records that intersect the surface between the two green vertical lines. The predicate "tt=8" returns records below the orange line. The surrogate predicate returns records whose $surID \in [4, 8]$. The result is

a set of records in the form of *(surID, $interval_{vt}$, $instant_{tt}$)* and is equal to {*(5, [1,5[, 1), (6,[3,7[,3), (7, [2,6], 6), (2, [8,12], 7)* }.

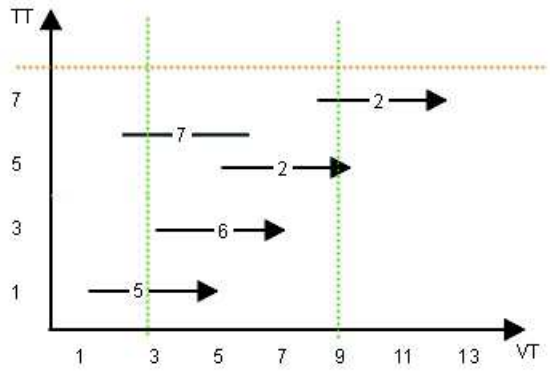


FIGURE 2.8: An attribute-column's entries

2.3.6 Physical Data Structures

ADI's physical structure is inspired by the Cassandra system [31]: data from each column is stored in three distinct data structures: *Alive Structure*, *memtable* and *SSTable* (Figure 2.9).

2.3.6.1 Alive

It is an in-memory data structure that contains live data, i.e newly added data with $vt_e = \infty$ that is likely to be modified. Once this data is *closed*, i.e the vt_e is set to a finite value, it is supposed that it will not be modified anymore, and is moved to the *memtable*.

2.3.6.2 Memtable

It is an in-memory structure that contains data that have been recently closed. Once the current memtable reaches a size threshold, it does no longer receive new data. A new current memtable is created and starts to receive data. The former *current memtable* becomes a *pending memtable*, and is asynchronously *flushed* as *SSTables* on disk.

2.3.6.3 Sorted String Table (SSTable)

It is an immutable file stored on disk that contains serialized columns. Since an instance can be modified and a SSTable is immutable, then information related to one instance can be spread over several SSTables.
Some queries can require the access to several *SSTables* to build this result. The fragmentation of column data over several disk files can badly impact their execution time. In order to avoid that, the column-store periodically merges SSTables into bigger size ones and rebuilds the indexes. If this approach induces a CPU overhead, it enables to reduce query execution time. The tasks of flushing memtables on disk and merging SSTables are asynchronous to data insertion task in order to avoid to slow it down.

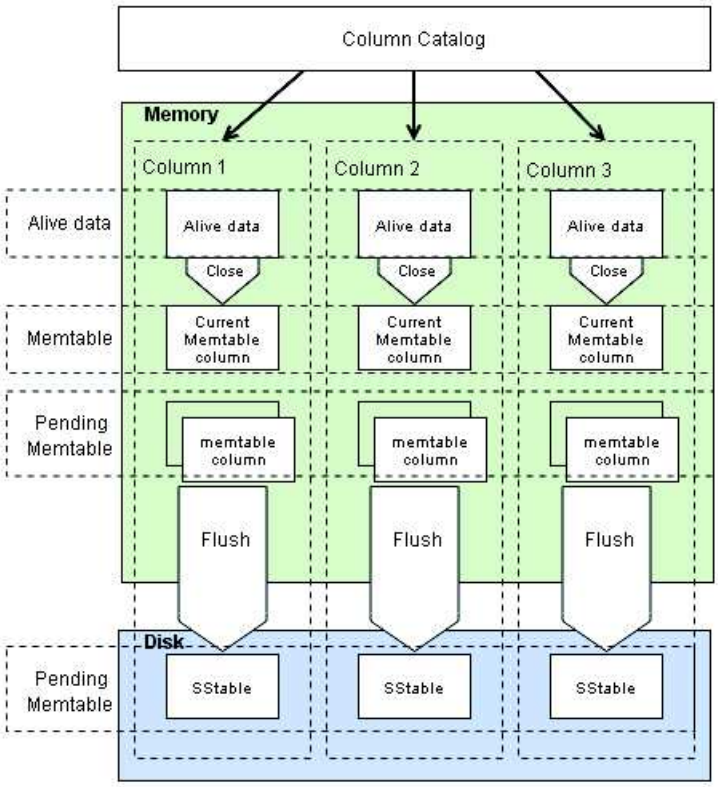


FIGURE 2.9: Physical data structure used by ADI

2.4 Alternative Temporal Modeling Approaches

In addition to the different temporal modeling approaches adopted in ADI, there are alternative approaches that we detail in this section, e.g the temporal enhancement of

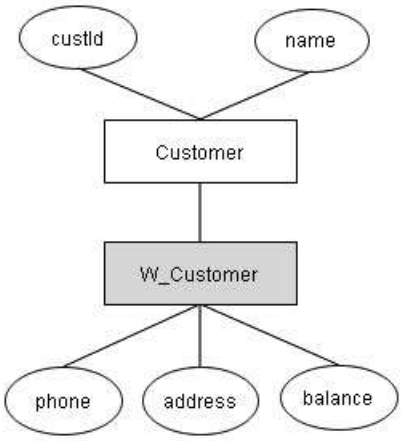


FIGURE 2.10: STEER diagram

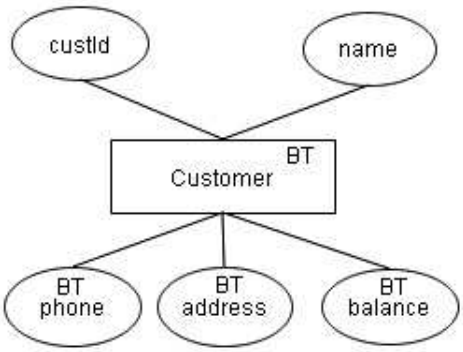


FIGURE 2.11: timeER Plus diagram

the ER model.

2.4.1 ER Model Enhancement With New Constructs/Annotations

There is another approach to handle temporality in the ER model. It consists in enhancing the ER model with new syntactical constructs or annotations that catch the temporal semantics. STEER [32], TERM [33] are examples of models that enhance the ER model with additional constructs based on two approaches to extend the ER. 1) One approach consists in adding new annotations to ER constructs that catch time semantics.2) The other approach consists in adding new constructs to the model that catch time semantics. In order to exemplify these two approaches, let us consider that the entity-type *customer* is bi-temporal as well as the attributes phone, address and balance. We suppose that the attributes *custId* and *name* are not temporal. This can be for example justified by the fact that it is not relevant to keep the history of these attributes or that they are not supposed to evolve through the two dimensions. Figure 2.10 is an example of using the STEER model to represent the entity-type *customer*. The approach consists in adding an additional construct (in grey) that is linked to the entity-type's temporal attributes. Figure 2.11 is an example of using the TimeER Plus model [34] to represent the same entity-time, by adding the annotations "VT" (valid time), "LT" (lifetime which is the valid time for entity-types) and "TT" (transaction time).

2.4.2 Tuple-Timestamping Approach

In addition to the *attribute-versioning* approach, there is also the *tuple-timestamping* approach. It consists in extending the conventional relation schema with temporal attributes which avoids any need to redefine the existing attribute. The storage overhead might be limited because the temporal attributes concern the whole tuple. However, it may introduce data redundancy because attribute values change at different times [35].

Besides being a *state-based* modeling approach, the Snodgrass model (Figure 2.7) which we introduced in the sub-section 2.3.3 is also a *tuple-timestamping* modeling approach. The first tuple indicates that AxTech's balance has the value 10 starting with the timestamp 1 and is still true, and recorded at 1 and logically deleted at 5. The second tuple indicates that the balance has the value 10 starting with the timestamp 1 to 3, recorded at 5 and is still current in the DB.

To the best of our knowledge, this is the most common approach among DBMSs with temporal capabilities whether they are *row-oriented* such as Oracle or *column-oriented* such as SAP Hana [36, 37].

2.4.3 Event-Based Modeling Approach

In addition to the *state-based modeling* approach that is adopted by ADI, there is an alternative approach, although less widespread, which is the *event-based modeling*. Jensen has proposed an event-based relational model [38, 39]. It consists in defining backlog relations, i.e append-only relations that do not authorize tuple update.
In this model, a backlog relation schema R is defined as follows:

$$\mathrm{R} = (A_1, A_2, \ldots, A_n, V_s, V_e, T, \mathrm{Op})$$

A_1, A_2, ..., A_n are non temporal attributes. The attributes V_s and V_e stores the bounds of the vt interval. Attribute T stores the transaction time when the tuple was inserted into the relation. Attribute Op indicates whether the tuple is inserted ("I") or deleted ("D"). A tuple that is inserted in the relation corresponds to an occurred event. It is current in the relation until a matching deletion tuple with the same explicit and valid attribute values is recorded. Concerning modifications, they are recorded by a pair of a deletion tuple and insertion tuple with the same T value. Figure 2.12 is an example of

such a relation. The first and the second tuples are examples of insertion tuples while the second and the third tuples are a modification pair. The *event-based modeling* approach

Customer	Balance	Nation	V_s	V_e	T	Op
AxTech	10	Spain	1	∞	1	I
Webtech	9	Germany	1	6	2	I
Webtech	9	Germany	1	6	3	D
Webtech	9	Germany	1	2	3	I
Webtech	12	Germany	3	6	3	I
Azeco	9	France	1	6	3	I
Azeco	9	France	1	6	4	D
AxTech	10	Spain	1	∞	5	D
AxTech	10	Spain	1	3	5	I
AxTech	30	Spain	3	∞	5	I
AxTech	30	Spain	3	∞	6	D
AxTech	30	France	3	∞	6	D
Jean Martin	9	France	4	∞	8	I

FIGURE 2.12: Example of Jensen's Backlog-Based relation

is adapted to some particular use cases such as when the database is append-only, i.e that once data is inserted into the DB, it can not be changed.

2.4.4 Temporal Generalization, Temporal Specialization

2.4.4.1 Temporal Specialization

The valid time and the transaction time are usually considered to be orthogonal [40], Usually there is no relationship between the valid time and the transaction time of any fact in the DB. However in many practical applications there is a restriction relationship between them. It is then possible to represent only one temporal aspect while the other one can be deduced. This is what we call a *temporal specialization* [41]. The authors defined 15 classes of specialization. One of them is the *degeneration*: a tuple's valid time is considered as valid when it is inserted into the DB. This means then vt and tt of all tuples are identical. In consequence, it is sufficient to store timestamps of one of the two temporal dimensions. Table 2.1b is an example of a *specialized relation* that is equivalent to the original relation in Table 2.1a, where we only keep the tt dimension.

Table 2.1: Specialization

(a) Before

customer_Name	*balance*	V_s	V_e	T_s	T_e
AxTech	250	15:00	16:00	15:00	16:00
AxTech	500	17:00	∞	17:00	∞
OneTwech	200	15:00	16:00	15:00	16:00
OneTwech	300	16:00	∞	15:00	∞

(b) After

customer_Name	*balance*	T_s	T_e
AxTech	250	15:00	16:00
AxTech	500	17:00	∞
OneTwech	200	15:00	16:00
OneTwech	300	16:00	∞

2.4.4.2 Temporal Generalization

If the temporal specialization aims to determine if there is any constraint between the valid time and the transaction time and thus coupling them, *temporal generalization* aims to decouple timestamps which enables to associate more than two temporal aspects to a fact. As an example let us consider the case of processing orders by a company. When an order is placed by a client, the company needs to check if it has the necessary funds for that order. If this occurs, then company accepts the order and it becomes effective. The order process concerns the reality, and thereby the valid time. The question is: "*what fact should we timestamp?*". Indeed several possibilities exist. If it is the fact "the order is placed", the valid time starts when the order is placed by the client and the transaction time is when that fact is stored in the DB. If we consider the fact "the order is processed", then it is valid when the company accepts the order, and the transaction time starts when that fact is stored in the database. We can see that the choice of the fact to store has an impact on the valid time and the transaction time. The question is how to represent to the best these facts. One possible approach would be to add an additional valid time timestamp to capture the fact that the decision to authorize an order is made. This particular timestamp is also called *decision-time* in the literature [42–45].

2.5 Chapter Synthesis

In this chapter, we introduced ADI, the Axway's OI platform, and its key component: the DBMS. ADI is a code-free platform that proposes a convenient GUI to implement and use OI applications since the majority of users have limited technical skills, e.g they can implement their data models thanks to the ER formalism. The choice has been made to not support a SQL-like language to implement queries, but rather an accessible API to the GUI. This API works in a declarative fashion (describes "what" and not "how"), and is mainly motivated by the fact that the DBMS is only intended to queries that can be implemented through the GUI.
ADI's DBMS is a NoSQL bi-temporal and column-oriented DBMS, inspired by *Cassandra*'s architecture, that has been specially designed to meet OI needs. Some choices have been made to handle the bi-temporality:

- *State-modeling* approach: Tuples store the state of data during a period of time.
- *Attribute-versionning* approach: Each attribute of the data model is timestamped with both one valid time and one transaction time dimension.
- *Generalization*, i.e handling more than one dimension per temporal aspect, as well as *Specialization*, i.e expressing one temporal aspect while the other can be deducted are not supported.

ADI's physical storage adopts a three-layers architecture where: 1) *Alive* and the *Memtable* are in-memory structures that respectively contains newly added data and future archived data. 2) *SSTable* is an immutable on-disk structure used to store archived data.

Chapter 3

Bi-temporal Database Benchmarking

3.1 Context

OI systems are critical due to their role to monitor their business. Such systems face several challenges: 1) They have to handle large volume of data, from fresh data to historical data. 2) They have to guarantee fast response times, so that users benefit from a "fluent" GUI. Therefore, it is crucial to be able to evaluate the performances of this kind of system. Since ADI is based on a bi-temporal database system, we consider benchmarks for bi-temporal databases as a good solution to measure the performances of our platform.

The widespread of temporal databases systems is mainly motivated by the need to record data evolution, e.g for auditing purposes or for making business decisions. Several vendors offer DBMS with temporal capabilities, e.g Oracle, SAP or Teradata. It appears then that comparing these systems is crucial to choose the appropriate system.

In this chapter we address the topic of benchmarking bi-temporal DBMS. It is structured as follows. First, we propose an overview of the main published works. Then we focus on a particular benchmark called TPC-BiH [2] which is, as far as we know, the most accomplished existing benchmark. Finally we propose an adaptation of that benchmark to meet our requirements.

3.2 Related Works

Benchmarking DBMS is an important topic addressed by the research community for years. Some benchmarks are references in the database domain: we quote the benchmarks proposed by TPC[1], covering the main database use cases. TPC-C and TPC-E are for example devoted to Online Transaction Processing (OLTP) use cases while TPC-H, TPC-DS and TPC-DI are designed for online Analytical Processing (OLAP) use cases.

Benchmarking temporal databases is not a new problem and many researchers have addressed it. In 1993, Jensen and al [46] proposed a functional benchmark that aims to asset the systems to support different classes of temporal queries. Unfortunately, their work does not include performance evaluation. In 1995, Duhman and al [47] proposed a framework to benchmark temporal databases. They provides a cookbook to implement a temporal benchmark, including requirements to build query workloads based on their use cases as well as requirements for implementing a temporal data generator. In 1998, Werstein [48] studied existing benchmarks at that time including TPC, the Wisconsin benchmark, BAPco, etc. He concluded that temporal aspects are not well supported.

In the last three years two performance benchmarks focusing on bi-temporal DB and based on the wildly used TPC's benchmarks have been proposed: One of them [49] was published at the VLDB TPCTC 2012 workshop[2]. The authors used the TPC-H benchmark, a benchmark devoted to decision support workloads, as a starting point and proposed a bi-temporal extension of it. They chose to extend a subset of relations - *part*, *supplier* and *partsupp*- with a two temporal attributes to express bi-temporality using Snodgrass modeling approach [50, 51]. This means that they adopt a tuple timestamping approach. They use data from TPC-H's data generator to initially populate the relations, then they use some functions to create the history for the three temporal relations. Concerning the query workload, they listed some possible queries that can be implemented. The second performance benchmark is introduced in the next section.

[1] http://www.tpc.org/
[2] http://www.tpc.org/tpctc/tpctc2012/default.asp

3.3 TPC-BiH

Kaufmann and al proposed a benchmark called TPC-BiH [2] and based on the TPC-H/TPC-C benchmarks, was published at the VLDB TPCTC 2013 workshop[3]. To the best of our knowledge, it is the most complete bi-temporal benchmark. Unlike in the previous benchmark, the data model is fully bi-temporal, i.e all relations are extended with both valid time and transaction time. It also contains a data generator that works in two steps. It first extends TPC-H data set with temporal data. Then it generates a history of data thanks to a workload of 9 business transactions. Finally TPC-BiH contains a workload of queries organized in 4 categories:*Pure-Time*, *Pure-Key*, *Range-Timeslice* and *Bi-temporal* queries.

3.3.1 Data Model

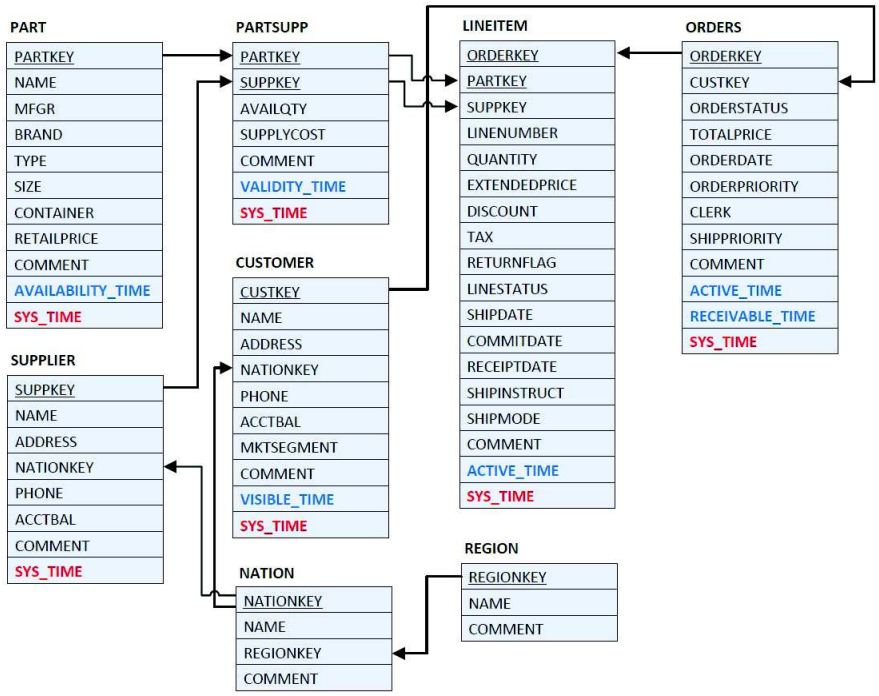


FIGURE 3.1: TPC-BiH's schema

The TPC-BiH's database schema is almost the TPC-H's database schema (see TPC-H's specifications on http://www.tpc.org/tpch/) with additional temporal attributes to express bi-temporality: each of them is an interval. Each of the relations *Part*, *partsupp*, *customer*, *lineitem* has one valid time attribute (e.g AVAILABILITY_TIME)

[3]http://www.tpc.org/tpctc/tpctc2013/default.asp

and one transaction time attribute. The relation *Supplier* is degenerated [13] since it is a particular specialization where the valid time and transaction time are identical. In consequence the relation contains only one temporal attribute which is the transaction time attribute (SYS_TIME). The relation *Orders* is a generalization because it contains two valid time attributes (ACTIVE_TIME and RECEIVABLE_TIME) in addition to a transaction time attribute (SYS_TIME). Finally the relations *Nation* and *Region* are not versioned and do not contain any additional temporal attributes. This is motivated by the fact that information concerning nations and regions are time invariant.

3.3.2 Data Generator

TPC-BiH contains a data generator that works in two steps. The first step consists in generating an initial data set using TPC-H. The generated data is stored as 8 files; each one containing one relation. Then the TPC-BiH data generator extends the initial data with temporal data generated using the TPC-H's temporal attributes. The second step consists in generating a history by using a set of 9 update functions (New Order, Cancel Order, Update Stock, ...). The generated data is stored in 8 files. Each file contains the complete history of one relation. In each file, the tuples generated during the first step of the TPC-BiH data generator are sorted according to the relation's primary key while the ones generated during the second step are sorted according to the transaction to which they belong.

We can notice that the TPC-BiH's model keeps some date attributes such as "SHIPDATE", "COMMITDATE" or "ORDERDATE" even if their information is supposed to be catch by the temporal attributes. We suppose that this choice is motivated by the desire to be backward compatible with TPC-H queries.

3.3.3 Queries

Introducing temporal dimensions expends the space of possible queries that can be expressed depending on how each time dimension is restricted: One can set both the valid time and transaction time to one instant or set one dimension to one instant while varying the other all over the time domain... TPC-BiH covers the query space by proposing a set of queries classified in four categories:

- *Pure-Time* query: It captures the state of the database at a specific time: a time dimension is fixed to a particular instant while the other one can either be fixed to an instant or vary over all the time domain.
- *Pure-Key* query: it addresses the issue of retrieving the history of a specific tuple: one time dimension is fixed to a specific instant while the other vary over all time domain.
- *Range-Timeslice* query : It is a bi-temporal extended version of a TPC-H query.
- *Bi-temporal* query [14]: It is a query that stress the two time dimensions in the same time.

3.4 Adaptation of TPC-BiH to ADI

The TPC-BiH can be seen as a general bi-temporal benchmark for decision support systems. Yet it needs to be adapted to be implemented on ADI:

- TPC-BiH adopts a *tuple-versionning* approach (subsection 2.4.2) to introduce time in the data model while ADI adopts an *attribute-versionning* approach.
- It uses both *generalization* and *specialization* modeling techniques which are not supported by ADI.
- TPC-BiH's authors do not address the DB populating strategy which may induce performance issues.

In this section, we first present the adapted data model. Then we present our strategy to populate the DB. Finally we present the workload of queries that suit to OI use cases.

3.4.1 Data Model

The schema we use in the benchmark is represented in (Figure 3.2) and is a little bit different from TPC-BiH's model (Figure 3.1). The generalization of "orders" by keeping two valid time dimensions "*active_time*" and "receivable_time" is replaced by another modeling approach. It consists in catching its semantic by adding a new possible value

"payable" to the attribute *orderStatus* that indicates that the order has been ordered but not paid yet.

We have removed the attributes "*commitdate*" and "*receiptdate*" of *lineitem* and the attribute "*orderDate*" of *orders* because they are redundant with the valid time dimension.

We use *Gadia's attribute value timestamped* representation that we have already introduced (subsection 2.4.2). In consequence, if we consider the relation symbol *Nation* with $schema(Nation) = \{nationkey, name, regionkey, comment\}$, then it would be defined as $schema(Nation) = \{\{(nationkey, T)\}, \{(name, T)\}, \{(regionkey, T), \{(comment, T)\}\}\}$ where $T = (vt, tt)$ with vt and tt respectively a valid time and transaction time interval. For shorthand, we use "value" to designate the value of the attribute, "vtb" and "vte" are the endpoints of vt, and "ttb" and "tte" are the endpoints of tt.

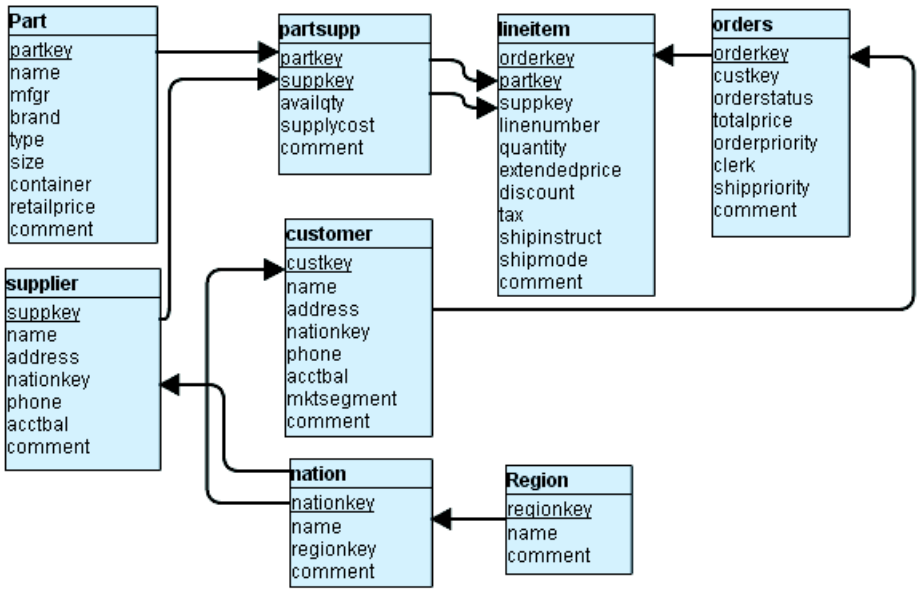


FIGURE 3.2: TPC-BiH's adapted schema

3.4.2 Database Population

From the initial data produced by the TPC-BiH data generator, we generate a stream of events *(id; data; T)*. Each event corresponds to an update order addressed to the database. "*Id*" is the event type, e.g "insert a new order" or "insert a new customer".

Data is the information handled by the event and T is the timestamps when the event occurred in the reality. The events are ordered according to the attribute T, so we can simulate a real-time workload.

3.4.3 Queries

TPC-BiH defines 4 classes of temporal queries that cover a large workload of queries. In the ADI case, we focus on a subpart of TPC-BiH's query workload that is relevant to us. It corresponds to implementable queries through ADI's GUI which are *Time Travel queries* and *Range-Timeslice queries*.

In ADI, queries are not specified using a SQL-like language. Instead, it proposes a convenient GUI process to guide analysts to specify their queries. Nevertheless, for the sake of simplicity, we use a pseudo-SQL formalism based on SQL:2011 [52] to express bi-temporal queries in this chapter as well as in the following ones. In the sequel, all aggregation queries use a one-day rhythm (01/01/1992, 1 day) represented by the relation r_day. It is defined over the relation symbol R_day with $schema(R\_day) = \{b, e\}$ where "b" and "e" are the attributes used to store respectively the beginning and the end of the rhythm's interval. For reasons of simplification, we assume that when we refer to an instant, e.g 01/01/1992, we mean 01/01/1992:00:00.

3.4.3.1 Time Travel Queries

A time travel query of this class returns the snapshot of a temporal database at a certain transaction time instant and a valid time instant. A time-travel query can either be a selection or an aggregation query. Listing 3.1 is an example of a *selection time travel* query. It considers the database at the most recent state (by default an SQL:2011 always returns the most recent data), and returns *orderkey*, *orderstatus* and *totalprice* of valid orders at the instant "01/01/1992". The predicate in the *where* clause (line 5) filters the orders that are valid at that instant (the instant 01/01/1992 must be in the interval [orderkey.vtb, orderkey.vte[).

```
1 SELECT orderkey.value,
2 orderstatus.value,
3 totalprice.value
```

```
4 FROM  Orders
5 WHERE orderkey.vtb<=01/01/1992 AND 01/01/1992<orderkey.vte;
```

Listing 3.1: Time travel query

An aggregation query is presented in Listing 3.2. It returns the number of orders grouped by customers that are processed by the company at the instant "01/01/1992" (line 4) considering the DB at the transaction time instant "01/01/1993" (line 3). Line 5 enables to get an order's customer while line 6 enables to get the customer's name that is valid at the order creation.

```
1 SELECT count(*)
2 FROM  Orders o, Customer c
3 WHERE o.orderkey.ttb<=01/01/1993 AND 01/01/1993<o.orderkey.tte AND
4 o.orderkey.vtb<= 01/01/1992 AND 01/01/1992<o.orderkey.vte AND
5 o.custkey.value == c.custkey.value AND
6 c.name.vtb<= o.orderkey.vtb AND  o.orderkey.vtb < c.name.vte
7 GROUP BY c.name.value;
```

Listing 3.2: Time travel aggregation query

3.4.3.2 Range-Timeslice Queries

Unlike the previous class of queries, these queries concern a range of time. In this case, the transaction time is always fixed to an instant while the valid time is fixed to an interval. This is motivated by the fact that OI applications are more interested in data evolution through the vt than the tt.

As for time-travel queries, there are two types of range-timeslice queries: a *selection query* and an *aggregation query*. Listing 3.3 is an example of the first type of range-timeslice queries. It considers the database at the most recent state, the query returns the history of all new orders that have been placed during the interval "[01/01/1991, 01/07/1991[". The predicates in the *where* clause filters the relevant orders

```
1 SELECT orderkey.value,
2 orderstatus.value,
3 totalprice.value
4 FROM  Orders
```

```
5 WHERE 01/01/1991 <= orderkey.vtb AND orderkey.vtb < 01/07/1991;
```

Listing 3.3: Range timeslice query

Concerning the *aggregation queries*, we follow Kaufmann and al's classification of time ranges in temporal aggregations [36]:

- *Instantaneous Aggregation*

Considering the database at "[tt_Instant]", an aggregation is performed at different instants spaced by the rhythm's period that belongs to a time interval "[vt_Interval]". As an example let us consider the query from Listing 3.4. Considering the most recent state of the DB, it computes for each day at midnight during the period "01/01/1992" to "01/07/1992" (one week) the number of orders that are still processed by the company, i.e the number of orders that intersect the instants {01/01, 01/02, 01/03, 01/04, 01/05, 01/06, 01/07, 01/08} (Figure 3.3). Line 3 enables to get the instants used to perform the aggregation (dots in Figure 3.3). Line 4 enables to get for each of these instants the orders that intersect it.

```
1 SELECT r_day.b,count(*)
2 FROM  Orders, r_day
3 WHERE '01/01/1992'<=r_day.b AND r_day.b<'01/07/1992'
4  orderkey.vtb<=r_day.b AND r_day.b<orderkey.vte
5 GROUP BY r_day.b;
```

Listing 3.4: Instantaneous Aggregation in range timeslice query

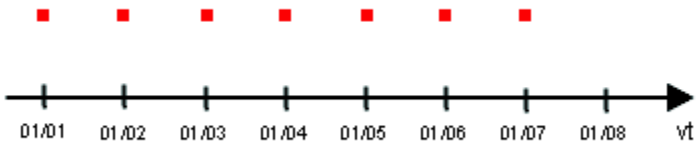


Figure 3.3: Instantaneous Aggregation

- *Tumbling Window*

In this case, the aggregation is performed on non-overlapping intervals, typically a rhythm's intervals. Listing 3.5 is an example of such aggregation type. Considering the

most recent state of the DB, the query returns for each day of the period "01/01/1992" to "01/07/1992" the number of new placed orders. Line 4 determines the aggregation intervals (Figure 3.4), while line 5 determines for each of them, the new orders.

```
1 SELECT r_day.b,r_day.e,count(*)
2 FROM  Orders, r_day
3 WHERE
4 '01/01/1992'<=r_day.b AND r_day.b<'01/07/1992' AND
5  r_day.b<=orderkey.vtb AND orderkey.vtb<r_day.e
6 GROUP BY r_day.b;
```

Listing 3.5: Tumbling Aggregation in range timeslice query

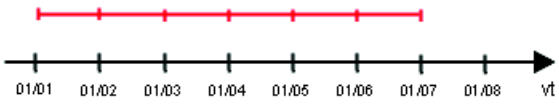


Figure 3.4: Tumbling window aggregation

- *Landmark Window*

The aggregation is performed on overlapping intervals that share the same interval beginning time instant. Listing 3.6 is an example of a such an aggregation. Considering the most recent state of the DB, the query computes the number of new placed orders for the intervals [01/01/1992, 01/02/1992[, [01/01/1992, 01/03/1992[, ..., [01/01/1992, 01/07/1992[(Figure 3.5). Line 4 determines the aggregation interval's endpoints while Line 5 determines the new orders per interval.

```
1 SELECT r_day.b,r_day.e,count(*)
2 FROM  Orders, r_day
3 WHERE
4 '01/01/1992'<=r_day.b AND r_day.b<'01/07/1992' AND
5  '01/01/1992'<=orderkey.vtb AND orderkey.vtb < r_day.e
6 GROUP BY r_day.e;
```

Listing 3.6: Landmark window Aggregation in range timeslice query

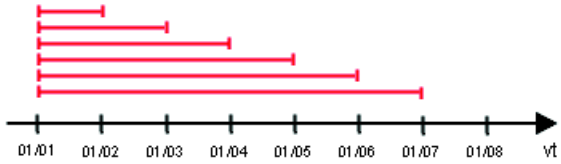


FIGURE 3.5: Landmark window aggregation

3.5 Experiments

We have conducted some experiments on both ADI's DBMS and a row-oriented DBMS with bi-temporal capabilities which we call *R-DBMS*. The objective is twofold: 1) to compare the performance of the two systems, and 2) to assess the impact of the database design on the system performance.

For the purpose of these experiments, we consider a sub-part of the TPC-BiH benchmark (Figure 3.6).

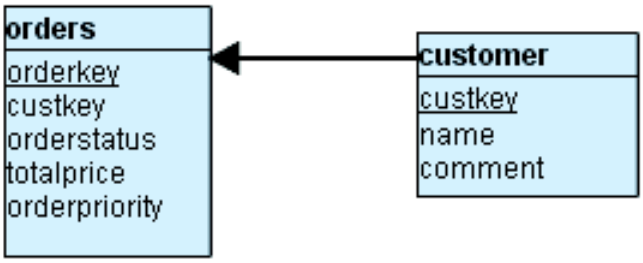


FIGURE 3.6: Conceptual Data Model

3.5.1 The Workload

We generate a stream of events corresponding to the period $[01/01/1992, 01/01/1993[$. There are two types of events:

- The **insertion** (I) consists in inserting a new order into the database. As we have limited the number of customers to 30, we do not consider their insertion.

- The **update** (U) consists in updating an existing orders in the database. The update concerns an order's status, its priority number or its ship priority.

Concerning the queries, we implemented four queries, each of them corresponds to one type we defined in the previous section, with some adaptations like: 1) the DB is considered at the most recent state. 2) We use an one-month rhythm (01/01/1992, 1 month) for aggregation queries instead of an one-day rhythm:

- Q_1: it is the *Time Travel selection* query defined in Listing 3.1. Instead of executing it considering the vt instant at "01/01/1992", we choose a random instant from the interval [01/01/1992, 01/01/1993[.
- Q_2: It is the *Time Travel aggregation* query defined in Listing 3.2. As for Q_1, we choose a random instant from the interval [01/01/1992, 01/01/1993[.
- Q_3: It is the *Instantaneous Aggregation* query defined in Listing 3.4 with a minor adaptation. It returns the current the number of currently processed orders at the begining of each month during the period [01/01/1992, 01/01/1993[.
- Q_4: It is the *Tumbling Window Aggregation* query defined in Listing 3.4. It returns the number of new orders per month during [01/01/1992, 01/01/1993[.
- Q_5: It is the *Landmark Window* query Listing 3.5. It returns the number of new orders each month since *01/01/1992*.

3.5.2 Logical Data Models

We choose to use three approaches of DB logical design in our testbed, so-called **model-T**, **model-C** and **model-M** in the sequel. The first two do not consider the temporal characteristics of the workload while the the third one does.

3.5.2.1 Model-T

In this model, each entity-type of the conceptual model becomes a relation schema. Each relation is timestamped with both the valid time and the transaction time.
The DB schema:

- Order(orderKey, totalprice, orderpriority, orderstatus, customer).
- Customer(custKey, name, comment).

3.5.2.2 Model-C

In this model, we translate the conceptual model according to an *attribute-versioning* and column-oriented target. Each entity type, relationship and attribute is stored in its own relation. Each relation is timestamped with both the valid time and the transaction time.
The DB schema:

- Order(orderKey)
- Order_totalprice(orderKey,totalprice)
- Order_priority(orderKey,orderpriority)
- Order_status(orderKey, orderstatus)
- Customer(custKey)
- Customer_name(custKey,name)
- Customer_comment(custKey,comment)
- ordered_by(orderKey, custKey)

3.5.2.3 Model-M

This approach considers the workload during the design by regrouping attributes that are not updated in the same relation. In consequence the attributes *status* and *priority* as well as the relationship *is_currently* have their own relations.

- Order(orderKey, totalprice)
- Order_priority(orderKey, priority)
- Order_status(orderKey, orderstatus)

- Customer(custKey, name, comment)
- ordered_by(orderKey, custKey)

3.5.3 Physical Data Model

Each of the three logical models is implemented in *R-DBMS*. Besides the three previous physical implementations, we also implement *Model-T* but without adding the temporal dimensions to relations. We denote that implementation "Model-WT" (WT stands for without time). The goal of "Model-WT" is to determine the overhead caused by the *R-DBMS* temporal features.

3.5.4 Results

In this section we present the results of 3 experiments we conducted to compare the performance of the different implementations we detailed in the previous section. The value of parameters used in the three experiments are detailed in Table 3.1. #I,#U and #Q are respectively the number of inserted orders, the number of updates and the number of queries. Experiments were executed on a virtual machine using VMWare. It runs on Windows 7 64 bits, is equipped with 12 GB of RAM and a Dual Core X5660 2.67Ghz.

Parameters	Experiment 1	Experiment 2	Experiment 3
Constants	#I =20k #Q = 0	#Q = 0	#Q = 2k
Variables	#U	#I #U = #I*10	#I #U = #I*10

TABLE 3.1: Experiment Parameters

3.5.4.1 Experiment 1: Update performance

The table 3.2 shows the size of the generated database for each implementation in the case of #*Q*=1000K. Concerning *R-DBMS*, we notice that the DB size in case of temporal implementations on *R-DBMS* (*model-T*, *model-C*, *model-M*), are very large compared to *model-WT*'s implementation and *ADI* by at least a factor of 10. It is due to the

complexity of the storage structure. Indeed, if the *model-WT* implementation contains 2 relations and 4 indexes, the *model-C* implementation contains 24 relations and 64 indexes (Table 3.3). We also notice that for each *R-DBMS* implementation, half of the

Implementations	Index size(Mo)	Table size(Mo)	Total(Mo)
ADI	10,5	42	52,5
Model-T	182	160	342
Model-C	198	185	383
Model-M	195	180	375
Model-WT	0,63	0,56	1,19

TABLE 3.2: Size of DBs in case of $\#U$=1000K

Implementations	Number of relations	Number of indexes
Model-WT	2	4
Model-T	9	24
Model-M	15	40
Model-C	24	64

TABLE 3.3: Number of data structures in the DB

storage structure size is occupied by indexes while it is barely 20% in the case of *ADI*.

The Figure 3.7 shows the time to execute the workload for each implementation. Concerning *R-DBMS*, the performance of the implementations are similar except for *Model-WT*. We suppose that the additional data structures used to handle bi-temporal data induce an overhead. As regards ADI, it outperforms *R-DBMS* temporal implementations. We suppose that, in addition to the data structure complexity of *R-DBMS* temporal, the main reason behind the performance gap is the *append-only* strategy adopted by ADI to store data on disk. This means that once a data is stored on disk, it can no longer be modified.

3.5.4.2 Experiment 2: Performance of Insert Operation

Experiment 2 aims also to evaluate the write performance of the implemented systems. Unlike the experiment 1, we choose here to vary the value of $\#I$ to generate 4 workloads. We also constrain the value of $\#U$ to $\#I * 10$.

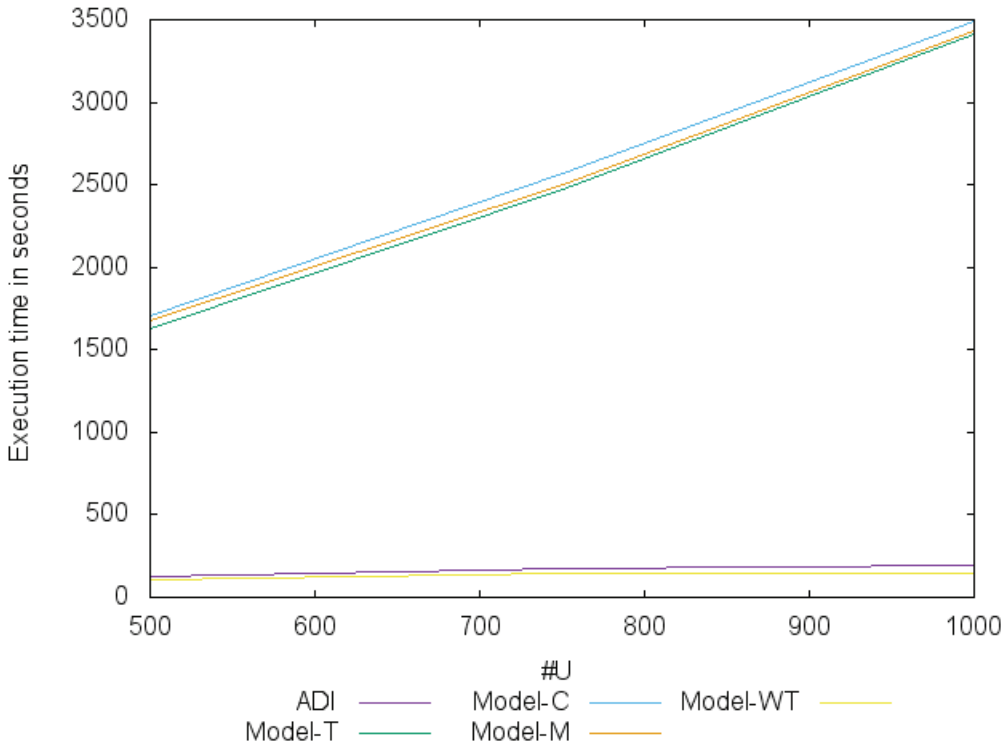


FIGURE 3.7: Experiment 1: Execution time of workloads

The table 3.8 shows the size of the generated DBs for each implementation in case of $\#I$=90K. Note also that the size of the DB in case of *model-T* is about 15% smaller than *model-M*'s DB and about 20% smaller than *Model-C*.

The Figure 3.9 confirms that ADI implementations have better performances than *R-DBMS* ones. Besides we can notice the difference in performances between temporal *R-DBMS* implementations. It is due to the order insert cost. Indeed inserting an order in the **Model-T** case consists in inserting one tuple in the relation *order*. In the **Model-C** and **Model-M** cases, inserting an order consists in inserting respectively five and four tuples in the DB.

Implementations	Index size(Mo)	Relation size(Mo)	Total(Mo)
ADI K	8	33	41
ADI HVC	10	26	36
Model-T	108	100	208
Model-C	127	128	255
Model-M	121	118	239
Model-WT	2,06	2,06	4,12

FIGURE 3.8: Size of the DBs in the case of $\#I$=50K

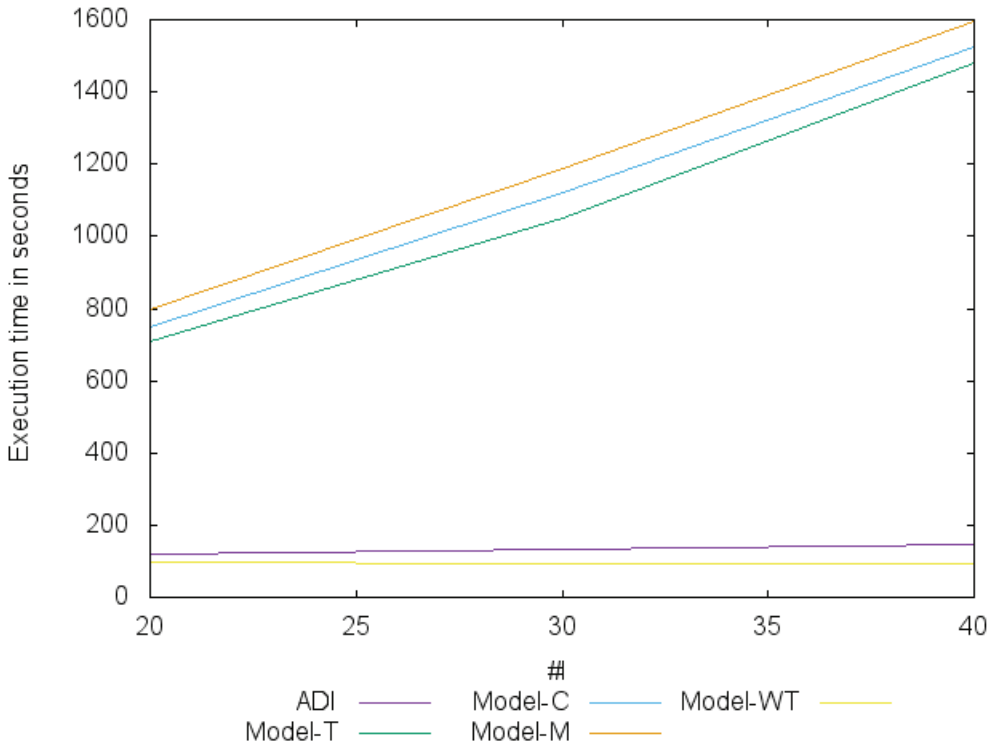


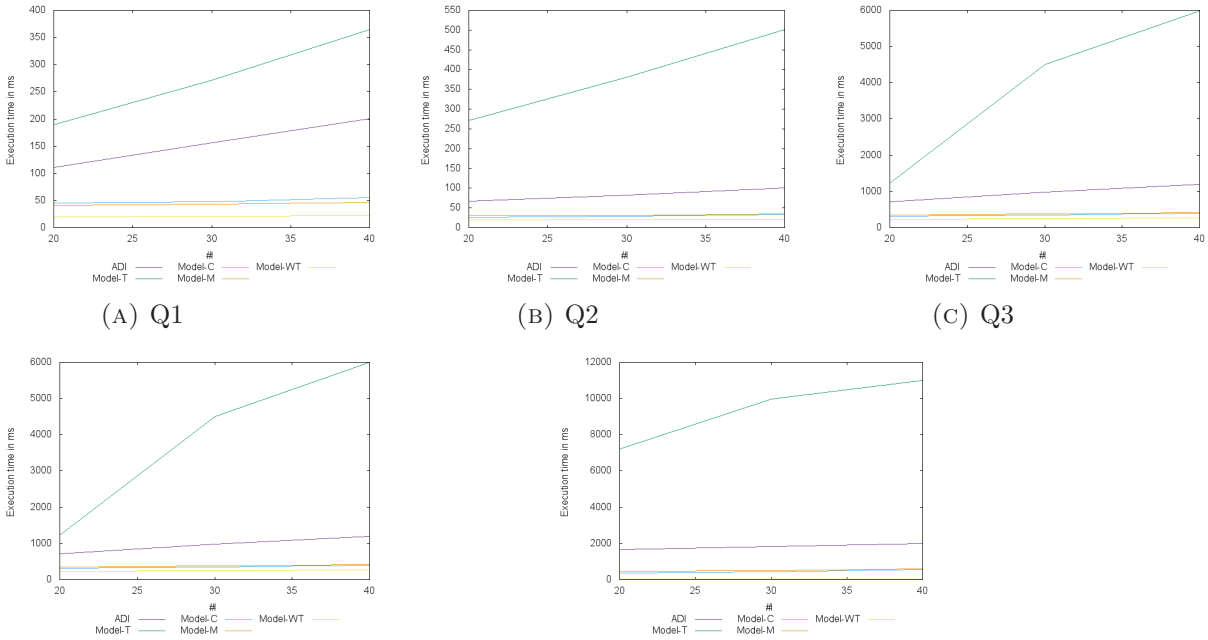
FIGURE 3.9: Experiment 2: Workload execution time

3.5.4.3 Experiment 3: Query execution performance

In this experiment we aim to evaluate the performances of ADI and *R-DBMS* in executing queries. As we suppose that DB size may have an impact on the query execution performances, we carry out four cases representing different DB sizes. The DB size is determined by using the workload parameters $\#I$ and $\#U$. We choose to vary the value of $\#I$ from 10k to 40k and choose to constrain $\#U$ to $\#I * 10$. For each value of $\#I$, we execute the query workload. The result is represented in Figure 3.10.

Concerning *R-DBMS* implementations, we can notice that *Model-T* has very poor performances compared to *Model-C* and *Model-M*. We suppose that *Model-T* bad performances are due to the fact that the update of *accbalance*, *phone* or the relationship *belongs_to* causes a data duplication in the relation *Customer* which decrease the query execution performance. However in the case of *Model-M* and *Model-C*, they are stored in their own relations and do not affect the query execution performance. It appears that both *Model-M* and *Model-C* have globally better performances than *ADI*. We assume that the indexes enable fast data access.

FIGURE 3.10: Experiment 3: Query execution time



(A) Q1 (B) Q2 (C) Q3

(D) Q4 (E) Q5

3.6 Chapter Synthesis

In this chapter, we presented a state of art of the bi-temporal benchmarks. To the best of our knowledge, TPC-BiH [2] is the most complete one. We proposed a new version of that benchmark that fulfills our requirements to assess ADI. Our main adaptations concern the modification of the data model as well as the DB populating strategy. We conducted some preliminary experiments with a twofold purpose: 1) Compare the performance of ADI with another DBMS called R-DBMS. 2) Asses different temporal database design strategies. The experiments learn us three important information: 1) ADI has higher data insertion speed than R-DBMS. We suppose that it is due to the fact that the first one adopts an *append-only* strategy to handle data on disk while and have simpler index structures. 2) In the cases when a good temporal database design is adopted, R-DBMS offers better query execution performances. In consequence some query optimization work has been conducted to enhance ADI's query execution performances. In the following chapters, we introduce two of them. 3) Handling bi-temporal data induces an important over cost.

Chapter 4

Aggregation Precomputing

4.1 Context

BI systems are usually used to generate non-interactive reports which do not have real-time (or near real-time) requirements. ADI however provides an interactive GUI that enables managers to monitor their business. It offers the possibility to explore real-time and other analyses computed over them. The underlying system must guarantee fast response time of queries in charge of feeding the GUI with information. GUI display lag makes the system unpractical.

In this chapter, we present a query processing optimization for ADI. It consists in precomputing query's aggregation operations as data is collected in order to reduce the GUI display lag.

This chapter is structured as follows. First, we introduce our query rewriting approach. After that, we address the issue of materialized continuous query computation scheduling. Then we point out how this approach has been implemented within ADI. The following section is devoted to experiments using the adapted version of TPC-Bih presented in chapter 3. Finally, we present some related works to the optimization, including materialized queries, data reduction, etc.

4.2 ADI Pre-computing Approach

Business Activity Monitoring systems usually provide managers with features to build so-called views[1] to monitor their business through user-friendly GUI. Those views use underlying queries to feed them with information to display. Consequently, they are not intended to be executed only once and then deleted. They can be evaluated several times, as long as the related view needs to be updated. In this section, we sketch the main idea of our query rewriting technique. Without loss of generality, we are concerned with the following class of temporal queries:

```
1 SELECT A1, A2, ..., An, Agg1, Agg2, ..., Aggk,
2   Rhythm_Relation.vtb,Rhythm_Relation.vte
3 FROM relation1, relation2, ..., relationJ,
4   stream1, stream2, ..., streamK,
5   Rhythm_Relation
6 WHERE tc1 AND tc2 AND ... AND tcn AND
7   c1 AND c2 AND ... AND cm
8 GROUP BY A1, A2, ..., An,
9   Rhythm_Relation.vtb, Rhythm_Relation.vte
```

Listing 4.1: Initial query Q_t

where:

- $A_1, A_2, \ldots, A_n$ are attributes or derived attributes,
- Agg_1, Agg_2, ..., Agg_k are aggregation functions, e.g., AVG, SUM, MIN.
- The WHERE clause is a conjunction of selection predicates and join predicates: *tcj* predicates are over temporal attributes while *ci* are over non temporal ones.
- $relation1$, $relation2$, ..., $relationJ$ are relations from the accessed database (historical data).
- $stream_1$, $stream_2$, ..., $stream_K$ are data streams (live data)
- *Rhythm_Relation* is the relation defined in the previous section.

[1]The term "view" is used here is used as a synonym of interface (GUI) which is different from the classical definition of "view" in DB

With ADI, such a query is used to feed an underlying GUI whenever it needs to be updated, e.g due to a user interaction. Whenever the amount of data to be processed exceeds some limits, GUI latency deteriorates. Therefore, to address the scalability issue, we rely on data reduction techniques [53]. Intuitively, we compute as soon as possible some partial answers allowing to efficiently answer a query asked by decision-makers. In other words, instead of performing aggregations at query time, we propose to perform them as soon as possible, when data arrives in the system. Thus, when a query is executed, it simply accesses the results of the aggregations which requires fewer I/O operations. This approach ensures that the most expensive I/O costs have been performed before the information is asked by a decision-maker. Hence, at query-time, the cost will be as low as possible, satisfying our major goal.
Given a bi-temporal query, we decompose this query as follows:

- one or more simple *continuous queries* (CQ) [54], and their results are materialized. Such queries handle large volumes of data and do not affect historical data. They are referred to as *materialized continuous queries*;
- one elaborated temporal query, referred to as an *on-demand query*, in charge of providing decision-makers with results is defined. Such a query accesses both historical and live data, including previously materialized CQ's results.

This approach has the advantage of providing a unified way to access both real-time and historical information through temporal queries. The result of this approach is *equivalent* to the result of the initial query against the same data. The reader is referred to [55] for equivalence of continuous queries. This is out of the scope of this document.

4.2.1 Materialized Continuous Queries

For each aggregation Agg_i in the initial query, we define one continuous query in charge of reducing input data into pre-computed aggregates. This query is simple and handle large volume of data, as in Listing 4.2.

```
1 SELECT A1,  A2, ..., An, Aggi, vtb, vte
2 FROM relation1, relation2, ..., relationJ,
3 stream1, stream2, ..., streamK, Rhythm_Relation
4 WHERE tc1 AND tc2 AND ... AND tcn AND
```

```
5    c1 AND c2 AND ... AND cm
6  GROUP BY A1,  A2, ..., An, Rhythm_Relation.vtb, Rhythm_Relation.vte
```

Listing 4.2: A materialized continuous query

where:

- *Aggi* is the aggregation operation performed by the query,
- $relation1, relation2, ..., relationJ$ is the set of accessed relations
- *vtb* and *vte* are two time attributes representing the time interval during which the computed result is valid,
- the result of this query is stored in a relation, thus becoming historical data.

Each continuous query is bound at its creation to a *rhythm*. For each interval of the rhythm, the query returns one result that is stored in the DB. The choice of the rhythm depends on the user's needs. The more accurate is the expected result, the finer is the rhythm's granularity, and the higher is the CPU cost and memory utilization.

Whenever a continuous query is created, some new attributes linked to that query are added dynamically to the database schema. This is intended to store the query results for future use. As an example, let us consider an instantaneous aggregation range time-slice query (Listing 4.3) that returns the total revenue of orders that are processed by the company every day at midnight during the period "[1/1/1992, 1/7/1992]" considering the most recent DB state. Lines 4 determines the aggregation instants, line 5 the relevant orders and lines 6 & 7 determine the order's totalprice value.

```
1  SELECT r_day.b,sum(totalPrice.value)
2  FROM  Orders o, r_day r
3  WHERE
4  "1/1/1992"<=r.vtb AND r.vtb<1/7/1992
5  o.orderkey.vtb=<r_day.vtb AND r.b<o.orderkey.vte AND
6  o.totalprice.vtb <= o. orderkey.vtb AND
7  o.orderkey.vtb < o.totalprice.vte
8  GROUP BY r.vtb;
```

Listing 4.3: Example of an instantaneous Aggregation range time-slice query

Our approach requires one continuous query to compute the total revenue per day (Listing. 4.4). We assume that the result is stored in an attribute called "totalRevenueInProcess" in a relation called "computations"

```
1 SELECT SUM(extendedPrice) as totalRevenueInProcess,
2   r.vtb, r.vte
3 FROM Orders o, r_day r
4 WHERE
5 o.orderkey.vtb<=r_day.vtb AND r.b<o.orderkey.vte AND
6 o.totalprice.vtb <= o. orderkey.vtb AND
7 o.orderkey.vtb < o.totalprice.vte
```

Listing 4.4: The continuous query to compute the total value of in-processed orders

4.2.2 On-demand Queries

An on-demand query is a bi-temporal query executed against the database whenever new information is required by decision-makers through their GUI.

```
1 SELECT A1, A2, ..., An
2 FROM relation1, relation2, ..., relationJ
3 WHERE tc1 AND tc2 AND ... AND tcn AND
4   c1 AND c2 AND ... AND cm;
```

Listing 4.5: On-demand query

According to the section 3.4.3, an *On-demand Query* can be one of the two following types: the *time travel*queries and the *time slice* queries.

If consider our example 4.3, then the underlying *on-demand query* would be Listing 4.6 where "totalRevenueInProcess" is the relation that contains

```
1 SELECT totalRevenueInProcess.vtb,
2 totalRevenueInProcess.vte,
3 totalRevenueInProcess.value
4 FROM  computations
5 WHERE
6 "01/01/1992" <= revenueInProcess.vtb AND
7 revenueInProcess.vtb<"01/07/1992"
```

Listing 4.6: Example of an on-demand query

4.2.3 Computation Scheduling of Materialized Continuous Queries

The use of materialized views requires to consider the scheduling strategy to compute its results. This strategy has to find a tradeoff between keeping views up-to-date as data is collected and limits the number of refreshes to reduce the computation cost. Instead computing a view for each single incoming update of entity, we prefer to refresh views periodically or by bunch of updates. The real-time aspect induces computation deadline constraints to ensure fresh information. The bi-temporality requires to consider the semantics of the two temporal dimensions to choose the adapted computation strategy. Indeed we have to determine the adequate instant when data is supposed to be available in the database to trigger the computation. In the general case the two dimensions are orthogonal, which means that there is no restrictions between the valid time and the transaction time of any fact in the DB. However in many practical applications there is a restriction relationship between them. For example, if we suppose that every event that occurs in the reality is considered as valid when it is inserted in the DB, then $vt_e = tt_e$. This topic has been addressed by Jensen and Snodgrass in bitemporal relational databases [41] under the name *temporal specialization relations*. The authors classify bitemporal relations into 15 classes of specialization.
In Decision Insight, we consider three types of events :

- *Retroactively bounded events*: It is the usual case. For each event, valid time and transaction time have the following interrelationships $0 < tt_e - vt_e \leq \Delta t$ with $\Delta t > 0$. In specific terms, the event occurs in reality at vt_e, then it is recorded in DB at tt_e. Δt is fixed by the user and represent the necessary time to collect it, transfer it to the DB and record it.

- *Delayed retroactive events*: It corresponds to events whose temporal attributes have the following interrelationships $\Delta t < tt_e - vt_e$. This type of events occurs in two cases: 1) when there is technical issue making difficult to deliver events to the DB. 2) to correct previous events that have been recorded into the DB.

- *Predictive events*: This case corresponds to events that are recorded into the DB before they occurs in reality ($tt_e \leq vt_e$), e.g a government tax rate modification which is always announced before it is applied so that concerned people make arrangements.

In order to handle these three types of events, ADI implements two different approaches: *Live Mode* and *Late Data Handler*.

- *Live Mode* : This approach is the usual mode and is in charge of handling both *retroactively bounded events* and *predictive events*. Concretely, considering a *materialized continuous query*, the condition to schedule its execution for an interval of its rhythm is that all input data are available. Thus, for a rhythm interval $[vt_{begin}, vt_{end}[$, the system supposes that at $tt = vt_{end} + \Delta t$ all input data is available and schedules the computation. Δt must smaller than $vt_{end} - vt_{begin}$. Otherwise, the computation task queue fill rate will be faster than the computation rate. In ADI, Δt is a platform parameter fixed by the user.
- *Late Data Handler*: This mode is dedicated to *retroactive events*. When such type of events arrives, the system determines all *materialized continuous query* and *rhythm intervals* impacted. Then it schedules their recomputation.

In the sequel, we restrict ourselves to the *live mode*.

4.3 Experiments

4.3.1 Database Populating

From the initial data produced by the TPC-BiH data generator, we generate a stream of events $< id, data, T >$. Each event corresponds to an updating instruction addressed to the database. *id* is the event type, e.g "insert a new order" or "insert a new customer". *data* is the information handled by the event and T is the timestamp when the event occurred. The events are ordered according to the attribute T, so we can simulate a real-time workload. The initial TPC-BiH dataset has a size of 400MB. The generated data stream contains 3620761 events (Table 4.1). We also introduce a scaling factor

Table 4.1: Number of operations per relation

Relation	# of insertions ($\#I$)	# of updates ($\#U$)	# of deletions (($\#D$))
Region	5	0	0
Nation	25	0	0
Supplier	1000	0	0
Part	20000	49861	0
Customer	164668	253430	0
Partsupp	80000	352391	0
Orders	348026	681103	8452
LineItems	939670	699310	22820

"SF" to fix the rate of the data stream. For the initial data stream $SF = 1$. All data streams with a higher SF are generated by duplicating SF times each event.

4.3.2 Queries

We have implemented two examples of typical queries used in BAM. Those queries are frequently executed by a GUI, requiring rapid response times.

4.3.2.1 Query 1

This first query, (Listing 4.7), aims at answering the following business question where "[YEAR]" is a parameter:

> "What is the sum of new revenues for the company every day from 1/1/1992 to 1/1/[YEAR] considering the most recent data?"

Lines 5 & 6 determine the rhythm intervals used to compute the aggregation. Lines 7 & 8 determine the new orders per interval while lines 9 & 10 determine the value of the orders. We redefine this query as one *materialized continuous query* "Q1-Cont" (Listing 4.8) and one *on-demand query* "Q1-OnD" (Listing 4.9).

```
1 SELECT Ryhthm_1d.vtb as vtb, Ryhthm_1d.vte as vte,
2   SUM(totalprice.value) as newRevenuePerDay
3 FROM Orders o, Ryhthm_1d r
4 WHERE
5 '01/01/1992'<=r.vtb AND
6 r.vtb < '01/01/[YEAR]' AND
7 r.vtb <= o.orderkey.vtb AND
```

```
8  o.orderkey.vtb < r.vte AND
9  o.totalprice.vtb <= o.orderkey.vtb AND
10 o.orderkey.vtb < o.totalprice.vte
11 GROUP BY r.vtb;
```

Listing 4.7: Q1: New Revenue per day

```
1  SELECT SUM(totalprice.value) as newRevenuePerDay,
2    r.vtb as vtb, r.vte as vte
3  FROM Orders o, Ryhthm_1d r
4  WHERE
5  r.vtb <= o.orderkey.vtb AND
6  o.orderkey.vtb < r.vte AND
7  o.totalprice.vtb <= o.orderkey.vtb AND
8  o.orderkey.vtb < o.totalprice.vte
```

Listing 4.8: Q1-Cont: New Revenue per day

```
1  SELECT vtb, vte, aggr as newRevenuePerDay
2  FROM NewRevenuePerDay n
3  WHERE
4  '01/01/1992' <= n.vtb AND
5  n.vtb < '01/01/[YEAR]';
```

Listing 4.9: Q1-OnD: New Revenue per day

4.3.2.2 Query 2

The query given in the (Listing 4.10) aims at answering the following business question where "[YEAR]" is a parameter:

> "What is the number of orders per status for every day at midnight from 1/1/1992 to 1/1/[YEAR] considering the most recent data?"

As for the Listing 4.7, the lines 5 & 6 determine the rhythm intervals for aggregation, the lines 7 & 8 determine the relevant orders per rhythm interval while the lines 9 & 10 determine their status. We redefine this query as one *materialized continuous query* "Q2-Cont" (Listing 4.11) and one *on-demand query* "Q2-OnD" (Listing 4.12).

```
SELECT r.vtb as vtb, r.vte as vte,
  COUNT(*) as numOrdersPerDayPerStatus
FROM Orders o, Ryhthm_1d r
WHERE
        '01/01/1992'=<r.vtb AND
        r.vtb < '01/01/[YEAR]' AND
  r.vtb <= o.orderkey.vtb AND
  o.orderkey.vtb < r.vte AND
        o.orderstatus.vtb <= o.orderkey.vtb AND
        o.orderkey.vtb < o.orderstatus.vte
GROUP BY r.vtb, o.orderstatus.value;
```

Listing 4.10: Q2: Number orders per status and per day

```
SELECT COUNT(*) as numOrdersPerDayPerStatus,
  r.vtb as vtb, r.vte as vte
FROM Orders o, Ryhthm_1d r
WHERE
        r.vtb <= o.orderkey.vtb AND
  o.orderkey.vtb < r.vte AND
        o.orderstatus.vtb <= o.orderkey.vtb  AND
        o.orderkey.vtb < o.orderstatus.vte
GROUP BY r.vtb, o.orderstatus.value;
```

Listing 4.11: Q2-Cont: Number of orders per status and per day

```
SELECT vtb, vte, numberOrdersPerDayPerStatus
FROM Computations
WHERE '01/01/1992' <= vtb AND
  vtb <'01/01/[YEAR]';
```

Listing 4.12: Q2-OnD: Number of orders per status and per day

4.3.3 Experimental Results

In this section we present the results of experiments conducted to assess the performances of our approach. To do this, we compare system performances with and without our optimization. We also show the overhead of our optimization. Experiments have been

executed on a physical machine which runs an Ubuntu 10.04, equipped with 12GB of RAM, an Intel i7 processor with 8 cores at 2.8GHz and a 4TB of RAID storage.

4.3.3.1 Response Time

In this test, we point out the interest of our approach in reducing ADI's response time. We run two experiments: in the first one, we fix SF to 1 and we evaluate the impact of the time range size on the execution time of both Q1 and Q2. In the second one , we fix the value of the parameter "YEAR" to 1996 and we vary the value of SF.

Fixed Scalar Factor

We inject a stream concerning the period [1/1/1992, 1/1/1999[. At the beginning of each new year of the simulation period, we execute once Q1 and Q2 using a new value of the parameter "[YEAR]". We compare two versions of each query: the optimized version, using our approach based on continuous queries (Q1-OnD and Q2-OnD), and a classical version, where the result is computed whenever the query arrives (Q1 and Q2). We collect the execution times of these queries and represent them on Figure 4.1. We

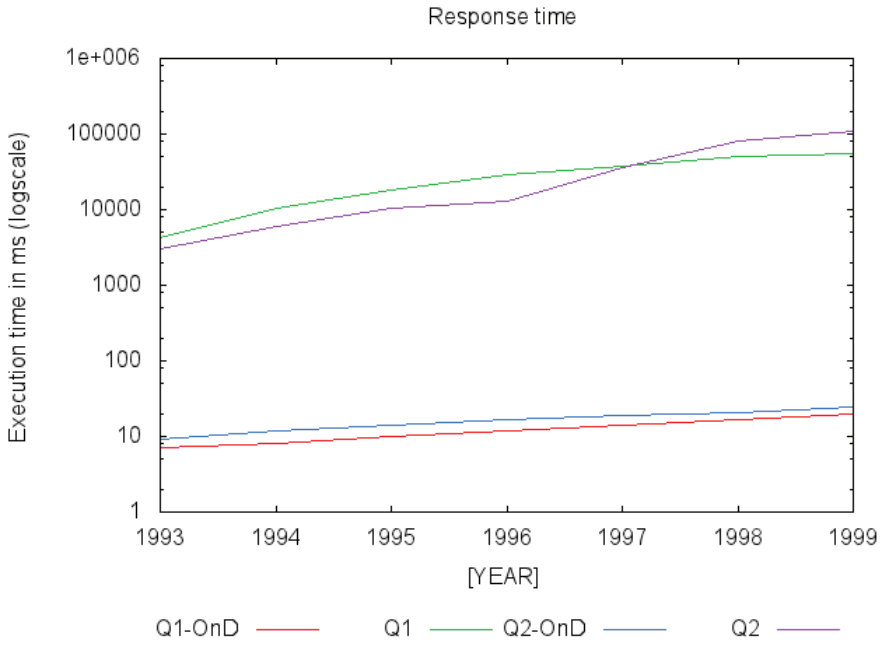


FIGURE 4.1: Query response time while varying window size

notice that optimized versions of queries outperform the rest by at least a factor of 100. For each day of the query interval, Q1-OnD accesses one value which is the materialized result of the underlying continuous query. Q1, however, accesses the original data, i.e

about 200 items for per day.

Varying Scalar Factor

In this experiment, we assess our approach when we vary the data stream rate. The experimental conditions are similar to the previous test. We vary the value of SF from 1 to 6. For each value of SF, we inject the stream that concerns the period [1/1/1992, 1/1/1996[. Following the injection, the queries Q1 and Q2 are successively executed with and without optimization. The queries are executed with [YEAR]=1996.

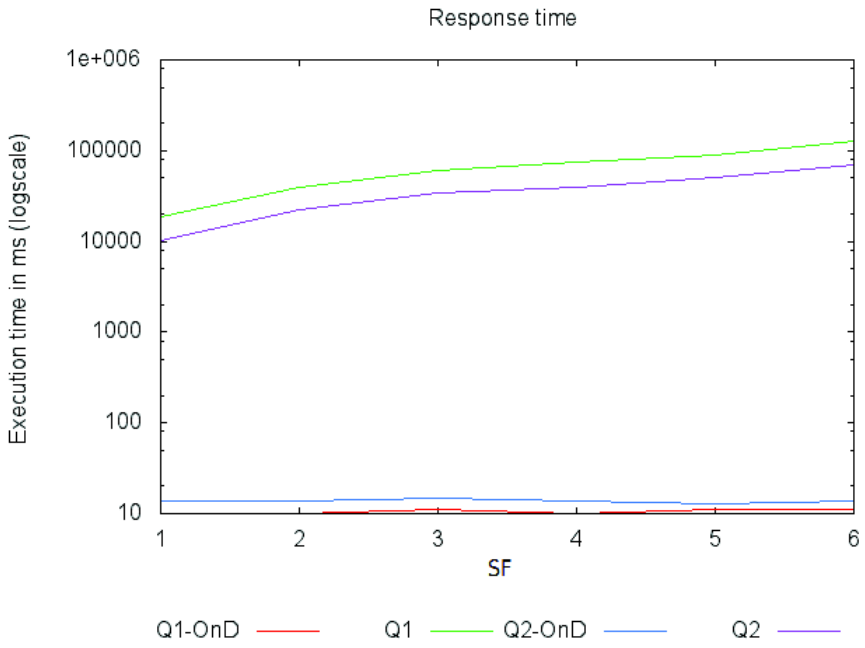


FIGURE 4.2: Query response time while varying data stream rate: YEAR=1996

When the data stream rate increases, the query execution time of the non-optimized queries increases, else it remains stable.

4.3.3.2 Precomputation Overhead

Previous tests demonstrate the advantage of our approach in reducing the response time of the system. However it induces a CPU and disk storage overhead.

Fixed Scalar Factor

The experimental conditions are similar to the test for response time/fixed scalar factor, except that we use only Q1. For each day of the simulated period, we collect the CPU

time of Q1-Cont. We also collect the CPU time to execute Q1-OnD and Q1. Fig.4.3 shows the results of the experiment: one curve represents the CPU consumption of Q1, while the other is the sum of the CPU consumption of Q1-cont and Q1-OnD.

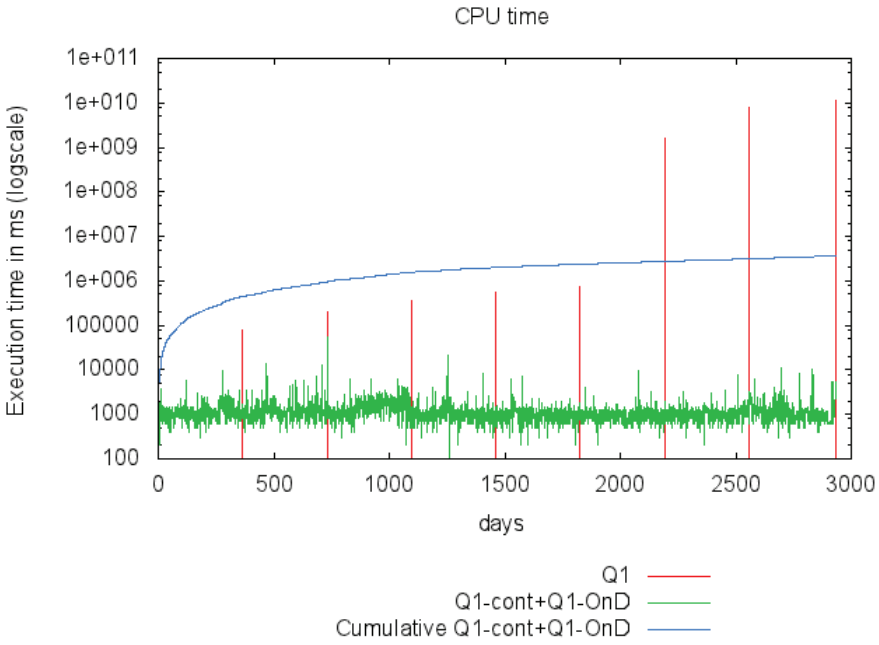


FIGURE 4.3: Continuous query computation overhead

It appears that the optimized approach requires a CPU overhead throughout the simulation time. However, it smooths the CPU consumption curve and avoids peaks at query time and thus system overload. We also notice that as from the 2000th day of simulation, the CPU Q1 cost is at least 100 times greater than the CPU required to compute Q1-cont and Q1-OnD. This means that for a query using a large time interval (6 years), the overhead induced by our approach has no impact on query processing performance.

Varying Scalar Factor

In this experiment we assess the cost of our approach as we vary the stream rate using the parameter SF. For each stream, we first inject the data stream corresponding to the period [1/1/1992, 1/1/1996[, then we execute Q1 with [YEAR]= 1996. We collect the CPU time to perform Q1 and Q1-OnD. We also collect the average CPU time of Q1-cont per day and the total sum of all CPU time consumption of Q1-cont during the simulation. The results are represented in (Figure 4.4).

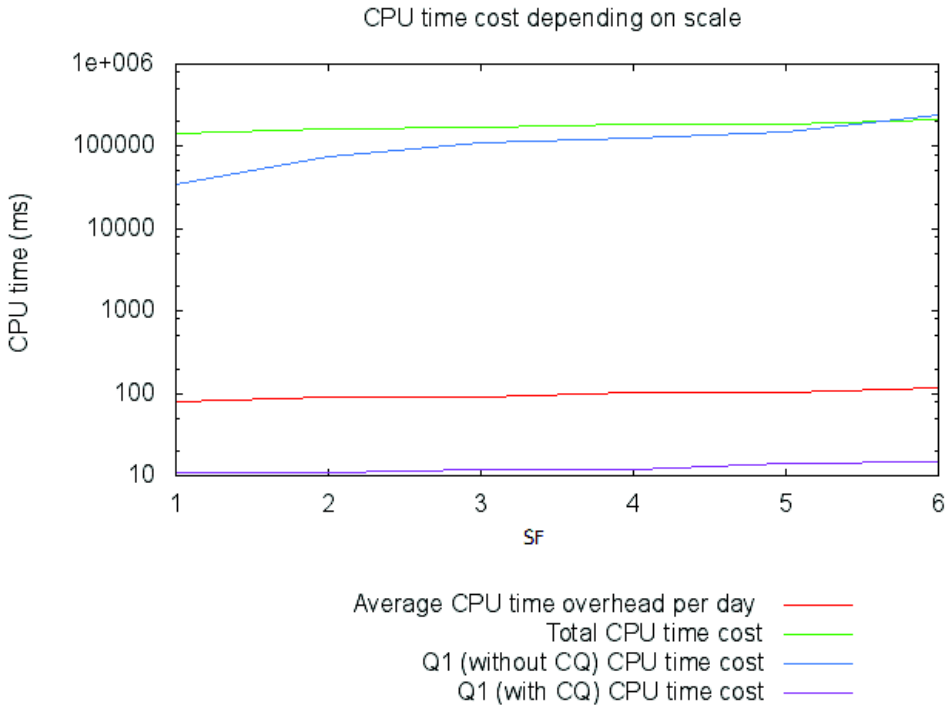


FIGURE 4.4: CPU time as a function of scale factor. YEAR= 1996

Whenever $SF >= 6$, our approach does not have any CPU overhead compared to the execution of Q1.

4.3.3.3 Concurrent Query Execution

In this test, we simulate several users interacting with the system. We have performed two experiments: one where we vary the number of concurrent queries and another where we vary SF for a given number of concurent queries (10).

Fixed Scalar Factor

In this experiment, we use a dataset where $SF = 1$. We first populate the system with data corresponding to the period [1/1/1992, 1/1/1999[. After data injection, we execute concurrently several instances of the query Q1 with [YEAR]=1999. Then we get the CPU time required to execute them all. Fig.4.5 shows the results of this experiment where we varied the number of simultaneous executed queries from 1 to 20.

As shown in Fig.4.5, our approach is quite adapted for execution of concurrent queries because it limits the CPU consumption.

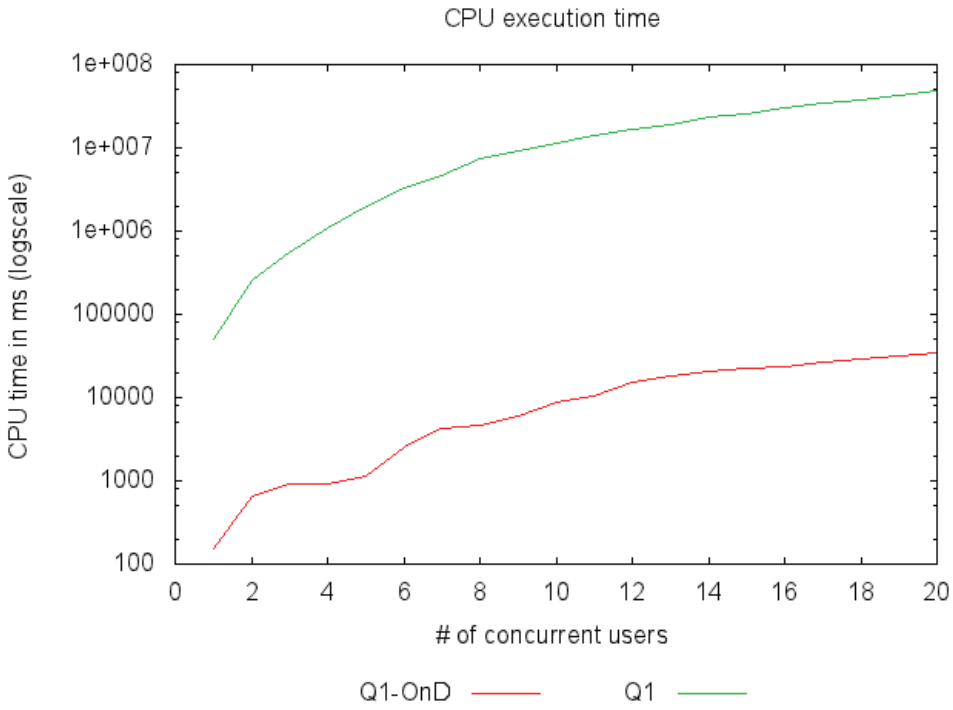


FIGURE 4.5: Concurrent query execution

Varying Scalar Factor

In this experiment, we explore the impact of the data stream throughput on the execution of concurrent queries. We first populate the system with data that corresponds to the period [1/1/1992, 1/1/1996[. Then we execute 10 concurrent queries, corresponding to 10 users.

As in the previous test, we observe the advantage of the proposed optimization as all the *on-demand* queries access the *continuous* queries' results while in the not optimized case, each query gets the original collected stream and computes the aggregation.

4.4 Related Works

Our optimization is at the crossroads of several topics. It includes query materialization, data reduction.

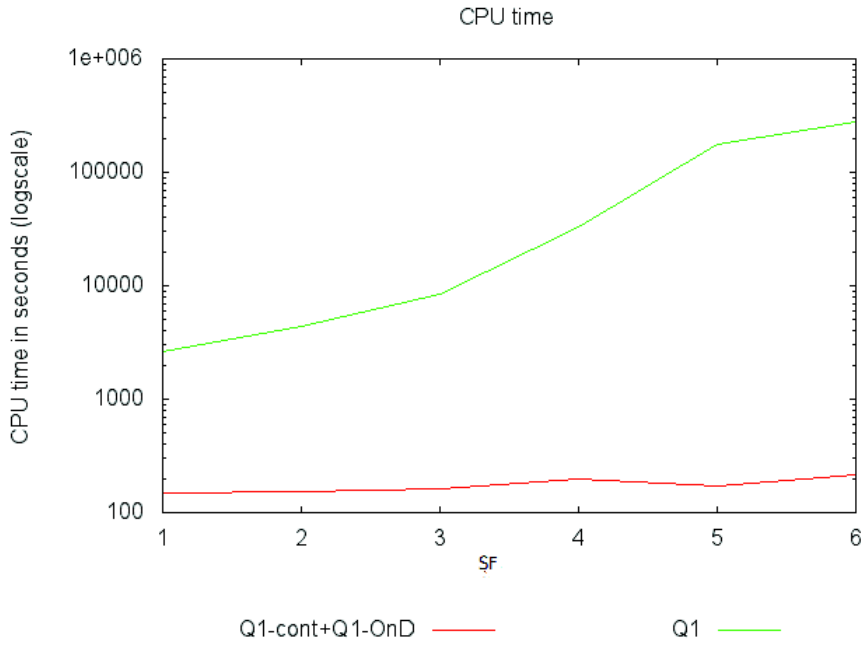


FIGURE 4.6: Concurrent query execution while varying SF

4.4.1 Postponing Query Processing

Postponing query processing when the system is at a lower load, e.g during the night, is not a new idea, see for instance [56] in a BI context. However this approach is not adapted to OI use cases because it is not intended to handle real-time data.

4.4.2 Load Shedding

When the input data of a DSMS exceeds its capacity, it is overloaded and its performance can be deteriorated. One existing approach consists in limiting the data rate at the system entrance to avoid any risk of system overloading [57]. The dropped data is lost for ever and can not be used anymore. This approach is adapted for real-time workload but not for historical workload.

4.4.3 Data Reduction

Data Reduction is the transformation of a large volume of data into a smaller one. There are many works in literature concerning data reduction techniques [53] which are widely used in the database systems to quickly get approximate answers from very

large databases. Traditionally they are used in two fields: *query optimization* and *data warehouses*.

The query optimizer needs to accurately evaluate the cost of alternative query plans to determine the "optimal" one to execute. Obviously the cost of searching an optimal plan has to be efficient and much less important than the query execution cost. Since the query execution plan cost depends on data characteristics, DBMS usually keeps statistics of the stored data which avoid costly accesses to data during the optimization process.

Data warehouses can be very large and thus querying them can take very long time. It appears that sometimes the user needs to have a quick global overview of the DB specially during the first phases of data exploration. Thus the answer speed is more critical than its accurate. In consequence data reduction can be used to return rapid results with a certain approximation.
Among these techniques, we can quote compression, sampling, aggregation,...

4.4.4 Combining Historical and Real-time Data

To the best of our knowledge, Chandrasekaran and Franklin were the first to address the topic of combining real-time data with historical data [58] in the academic field. They noted that the main performance issue for those systems was the I/O cost induced by gathering historical data, which decreases drastically live data stream processing performance. They proposed a framework using some data reduction techniques for historical data to limit I/O cost (see also [53]). Their framework data reduction level to be adapted with respect to current available resources. They defined three approaches to perform data reduction techniques: *OnWriteReplicate*, *OnReadModify* and *Hybdrid approach*.
The first approach is based on the fact that random disk I/Os are expensive. Data reduction is performed continuously as soon as data is collected by the system. Thus at query time, the global query can access pre-computed results when needed. Nevertheless, pre-computed results can never be accessed by global queries.
The second approach consists in performing the data reduction at query time only. The price to be paid can be very high for delivering timely information. The third approach combines the two previous approaches and shares the work between data arrival and

query time. In this approach there is a single copy of the stream stored on disk and divided into separate batches. Each batch is divided into a fixed number of blocks. Tuples are randomly inserted in different blocks of the current batch. Once one block is filled, the entire run is flushed on disk. At query time, the system only accesses a fraction of blocks of runs according to a sampling rate.

4.5 Chapter Synthesis

In this chapter, we proposed an optimization that is currently implemented in ADI and which reduces the GUI display lag. It consists in pre-computing aggregation at data arrival and materialize the results for future uses.

This optimization assumes there is a restrictive relationship between the vt and the tt of an event. Indeed, an event that occurs at the instant vt_e will be inserted in the DB at the instant tt_e such that $0 < tt_e - vt_e \leq \Delta t$ with $\Delta t > 0$.

The experiments we conduced, using the adapted version of TPC-BiH presented in Chapter 3, showed that ADI is able to deliver very fast responses with acceptable CPU over cost.

Chapter 5

Cost Based Optimizer

A query is transformed into an *execution plan* which is executed by the DBMS. The classical process consists of the following steps. First the query is *parsed* in order to check if it is well formulated according to the query language's syntax rules. It is also validated by checking if all attribute and relation names exist in the target DB. This query is then transformed into an internal data structure that can either be a *query tree* or a *graph tree*. Next, the DBMS has to define the *execution plan* for retrieving the query's results. A query can have multiple executions plans. Its objective is to find the adequate one. This task is called *query optimization*. The term *optimization* is actually a misnomer because the generated *optimized plan* is not the absolute most efficient one, but rather the best one according to the optimization strategy. Optimizing a query execution plan is a quite tough task. The main challenge is to find an efficient way to determine an acceptable execution plan for a given query using the least possible resources. Indeed an expensive approach that finds optimal solutions can not be interesting considering the performance gains.

ADI's query engine is based on a similar process and transforms a query expressed withing a GUI into an execution plan. The current version of ADI does not embed a *query optimizer*.

In this chapter, we introduce an ongoing work to implement a *cost-based query optimizer* adapter for bi-temporal queries, the core of ADI's query engine. We first define the main topics that are involved in building a cost model optimizer. This includes the *search space*, the *optimization algorithm* and the *cost model*. If the first two ones does not

require taking into account the bi-temporal nature of the handled data, the third one does. Indeed, a good cost model requires for example to keep relevant statistics on data as well as good estimations of data access. Then we present the implementation of the optimizer in ADI. Finally we conclude the chapter with some experiments that assess the interest of the optimizer.

5.1 Problem Statement

There are several optimizations that can be considered while building an optimal execution plan. We can quote the choice of a suitable implementation for each operation, the form of the execution plan, the order of operations to execute, etc. Optimizations can be grouped into two main categories:

- The first one is based on *heuristic rules*, which consists in building a better execution plan using some common sense rules, e.g pushing top operators or avoiding access to the same information several times.
- The second category is based on *cost-based optimizations*, which consists in comparing the estimation cost of several execution plans and chooses the least expensive one.

In a first attempt to propose a query optimizer for ADI, we focus on determining the best order to execute join operations [59–62] This include tuple materialization operation [21], also called tuple reconstruction, which is specific to column-oriented DBMS. It consists in reassembling the attribute's columns that belong to the same *entity-type*. In our case, we assimilate it to a join operation.
An execution plan can be defined as a *tree* (Figure 5.1) where the leafs are column access operations (gets and scans cf 2.3.5) and the nodes are join operations. An enumeration of all possible plans is unpractical, and some heuristics have to be applied to reduce the search space.

This approach requires to consider three topics: the *search space*, the *cost model* and the *optimization algorithm* We detail them in the next section.

5.2 Related Work

5.2.1 Cost Model

The first proposal of a cost model based query optimized was introduced in the DBMS *System R* [60] in 1979. Since then, there were an important research activity concerning this topic to determine an accurate cost for a query during the optimization process. A cost model relies on three different components:

- *Data access cost* which includes for example data search, data block loading, network transfer, etc.
- *Function processing cost* like sorting, aggregation, etc.
- The *size of results* generated by different execution plan operators.

The first two components are rather related to the physical storage characteristics as well as on algorithms used to handle data. These two components are common to all DBMS types including bi-temporal ones, the third is related to data characteristics.

To the best of our knowledge, there are only two works that addressed the topic of estimating execution plan's intermediate results size for bi-temporal DB. 1) Segev and al [63] have proposed a set of simple formulas to estimate selection and joins assuming some data distribution hypothesis. 2) Slivinskas and al [64] have also proposed a cost model for their bi-temporal middleware with some techniques to estimate results of selections, joins, projections as well as aggregations. This includes the use of both histograms and simpler formulas.

5.2.1.1 Search Space

A *search space* is defined as the set of all *execution plans* that produce the same result. Each point of this space is a potential solution. The goal of the optimization is to find the point in the solution space with the lowest cost. The point is that the combinatorial explosion makes the exhaustive path of the whole space impossible. In consequence, heuristics are usually used to reduce the search space. Solution trees can be of different forms: *left-deep tree* (Figure 5.1-(a)), *right-deep tree* (Figure 5.1-(b)), *zig-zag tree* (Figure

5.1-(c)) and *bushy tree* (Figure 5.1-(d)). The difference between a *right-deep tree* and a *left-deep tree* is that in the first one all transient relations are consumed in pipeline while in the second one they are stored. *Bushy tree* is simply a tree that not match any of the three other forms. *Right-deep tree* and *zig-zag trees* are used in distributed computing environments [65]. Steinbrunn and al [59] addressed the topic of reducing

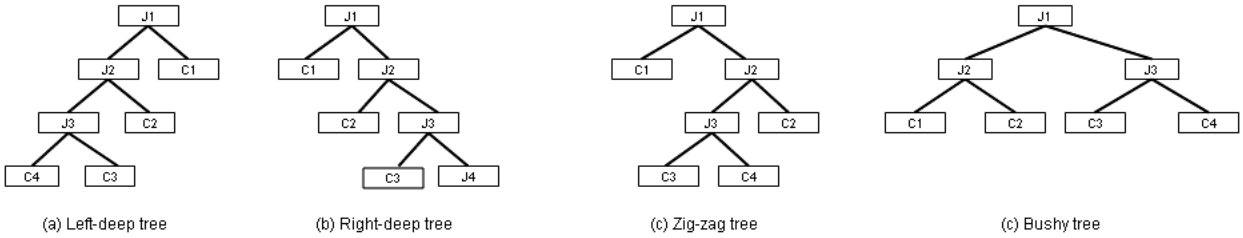


FIGURE 5.1: Different types of tree queries

the search space. The idea is to determine the cases where an optimal solution from the *left-deep tree* space search has great chances to be the global optimal solution of the whole *bushy-tree* solution space.

5.2.1.2 Optimization Algorithm

There are several strategies to explore the solution space that have been surveyed by Steinbrunn and al [59]. They can be divided into 4 classes.

- *Deterministic Algorithms*

  A deterministic algorithm builds a solution in a deterministic way using an heuristic or an exhaustive search. There are several algorithms of this class. We can quote the *dynamic programming* approach (Algorithm 1) which is historically the first algorithm used to optimize query plan in System-R [60]. The main disadvantage of this algorithm is its expensive consumption due to the generation of partial solutions. In consequence it becomes very expensive to apply this algorithm for queries with more than 10 relations to join. A more efficient variant of the dynamic programming approach has been proposed by Vance and Maier [61] that enables to efficiently handle queries with up to 18 relations. There are also other algorithms such as *Krishnamurthy-Boral-Zaniolo* (KBZ) [62] and the *AB* [66] algorithms.

Algorithm 1 Dynamic programming algorithm

```
1: Input:relations
2: Output:optPlan
3: partialSolutions := getAttributeAccesses();
4: //return the set of the scan accesses of all involved attributes
5: for (int i=1;r <|relations |;i++) do
6:     for all execPlan ∈ partialSolutions do
7:         for all r ∈ relations do
8:             if !(r ∈ optPlan) then
9:                 optPlan := optPlan + r;
10:            end if
11:        end for
12:    end for
13:    clean(partialSolutions);
14:    //remove all elements with equivalent and optimal alternative.
15: end for
```

- *Randomized Algorithms* The solutions are seen as points in a space, which are connected by edges. Each edge can be seen as a *move*, i.e a transformation of a solution to another one according to some rules. The algorithms perform a random walk through the solution space. The optimization ends once there is no more possible authorized move by rules or all moves have been consumed.
  Swap algorithm [67], for example, exchanges the position of two relations, while 3Cycle [68] performs a cyclic rotation of 3 relations.

- *Genetic Algorithms* [69]
  A set of initial random population of solutions is used to produce a new generation of members using genetic techniques such as random crossover and mutations. The best members, according to a cost function, survive to the next generation. The process ends once there is no more possible improvement or after reaching a predetermined number of generations.

- *Hybrid Algorithms*
  Hybrid algorithms combine deterministic approaches and randomized or genetic approaches. The solutions obtained by using deterministic algorithms are used as a starting point for genetic or randomized algorithms.

5.3 Our Approach

In a first attempt to propose a query optimizer for ADI, we focus on a particular optimization called "Join Ordering". It consists in determining the "optimal" order to execute execution plan's join operations. In our approach, the join operation includes the tuple materialization operation [21], also called tuple reconstruction, which is specific to column-oriented DBMS. It consists in reassembling the attribute'columns that belong to the same *entity-type*.

In this context, we define an execution plan as a *tree* where the nodes are join operations and the leafs are scans (subsection 2.3.5). We do not consider the get operations because they return at most a result of cardinality of 1. We choose to consider only *left-deep tree* execution plan which limits the space search. Figure 5.2 gives a simple query's execution plan that join the attributes "*name*", "*custKey*" and "*balance*" of the relation "*Customers*".

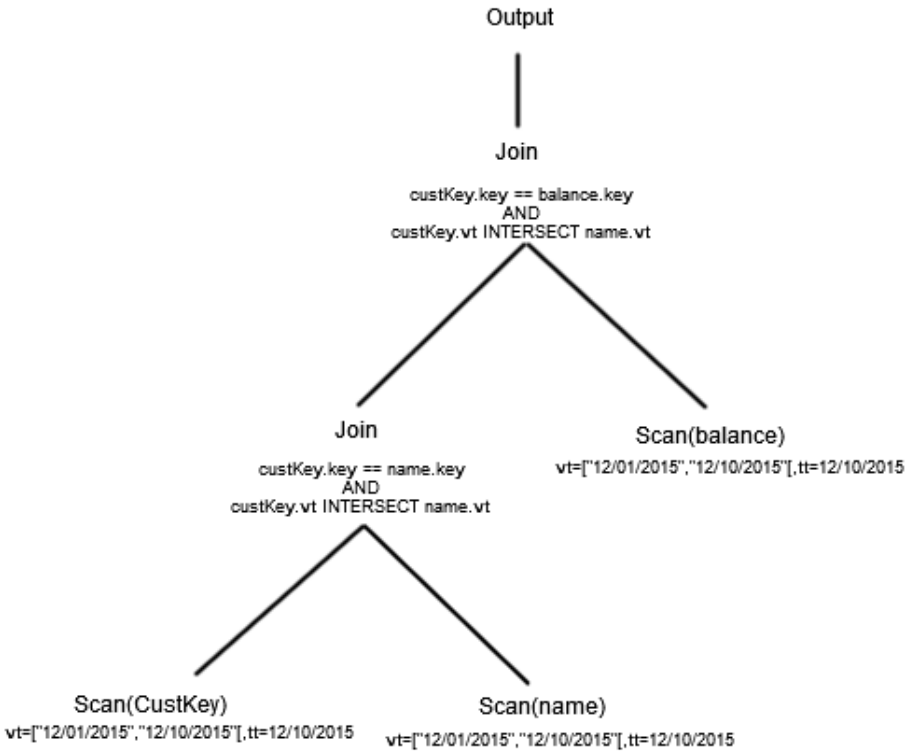


FIGURE 5.2: A simple ADI execution plan

To compare an execution plan cost, we use a *cost model* that is only based on estimating the size of results generated by the execution plan's nodes. An execution plan's cost

is simply the sum of its node's result estimation size. We assume that the bigger the node's results are the more expensive the execution plan is.

In this section, we define ADI's execution plan's operators: the *scan* and the *join* and present our approach to estimate the results generated by these operators. Then, we introduce our optimization algorithm. Finally we detail how the optimizer is implemented within ADI.

5.3.1 Column Scan Estimation

The scan of a column C can be defined as $\sigma_{p_{vt} \wedge p_{tt} \wedge p_{surID} \wedge p_v}(C)$ where p_{vt}, p_{tt}, p_{surID} and p_v are respectively the valid time, transaction time, surrogate attribute and the value predicates. We consider two assumptions:

- Data distribution does not depend on the transaction time. In consequence, the transaction time predicate, p_{tt}, is not considered in our estimation.
- The 3 dimensions (vt, surrogate and value) are independents, which means that the distribution of data according to one dimension does not depend on the others.

Concerning the surrogate and the value dimensions, we adopt the classical approach. For instance, we maintain for surrogate field of a column a *width-balanced histogram*. In the sequel, we focus on the vt dimension.

The objective is to estimate the cardinality of a set of intervals of a column C that intersect a given interval $[I_b, I_e]$. To do this, we use a formula proposed by Slivinskas and al [64] which we detail in the current section. It simply takes into account the trivial fact that the begin of an interval always precedes its end. Let us consider the functions $StartBefore(i, C)$ and $EndBefore(i, C)$ that return respectively the number of intervals from the column C that start and respectively end before the instant i. Then the cardinality of a set of intervals of C that intersect the interval $[I_b, I_e]$ can be estimated as $StartBefore(I_e, C) - EndBefore(I_b, C)$.

To compute the value of $StartBefore(i, C)$ and $EndBefore(i, C)$, we use two histograms H_b and H_e to store respectively the distribution of interval's begins and interval's ends. For a given histogram H, we define the following functions:

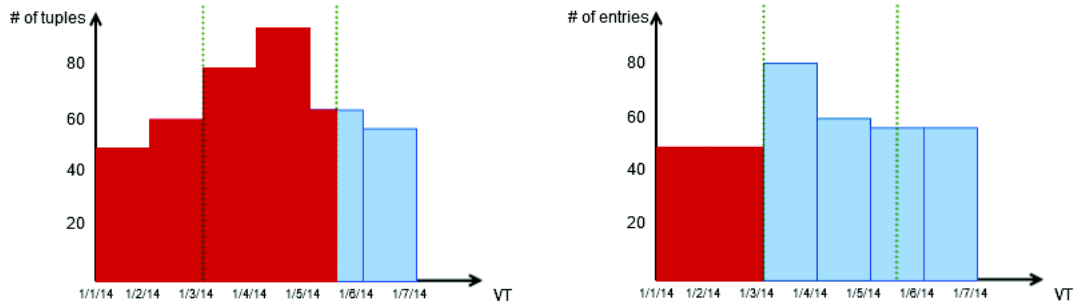
- $b_b(i, H)$ and $b_e(i, H)$ return respectively the start and the end of the i^{th} histogram bucket.
- $b_{val}(i, H)$ returns the value of the i^{th} bucket.
- $b_{No}(i, H)$ returns the bucket number that includes the instant "i".

The functions *StartBefore* (5.1) and *EndBefore* (5.2) are similar and are computed as follow. First the bucket containing the attribute *instant* is found. Then we sum the number values in all preceding buckets. We also add a fraction of the number of values in the bucket containing *instant*, assuming that values are uniformly distributed within the bucket. This approach is applicable for both *height-balanced histograms*, i.e where each bucket has the same number of values, and *width-balanced histograms*, i.e where each bucket is of the same length. In our case we adopt the second histogram type.

$$StartBefore(instant, C) = \sum_{i=1}^{b_{No}(instant,H_b)} (bVal(i, H_b)) + \frac{instant - b_b(b_{No}(instant, H_b), H_b)}{b_e(b_{No}(instant, H_b), H_b) - b_b(b_{No}(instant, H_b), H_b)} * b_{val}(bN_0(instant, H_b), H_b) \tag{5.1}$$

$$EndBefore(instant, C) = \sum_{i=1}^{b_{No}(instant,He)} (bval(i, He)) + \frac{instant - b_b(bN_0(instant, H_e), H_e)}{b_e(bN_0(instant, H_e), H_e) - b_b(bN_0(instant, H_e), H_e)} * b_{val}(bN_0(instant, H_e), H_e) \tag{5.2}$$

As an example of this approach, let us consider two width-balanced histograms H_b (Figure 5.3a) and H_e (Figure 5.3b) that represent respectively the distribution of interval's begins and ends of a column C. Let us estimate the result size of the scan of C with interval = [3/1/14T00:00:00, 6/01/14T12:00:00]. Then the estimation of the scan will be StartBefore(interval.end, H_e)-EndBefore(interval.begin,H_b). According to the Figure 5.3a, StartBefore(interval.end,H_e) $= 50 + 60 + 80 + 90 + \frac{1}{2} * 60 = 310$ (red buckets) and EndBefore(interval.begin,H_b) $= 50 + 50 = 100$ (red buckets). Thus, the estimation of the scan is equal to $310 - 100 = 210$.



(A) H_b (B) H_e

FIGURE 5.3: vt's histograms

5.3.2 Join Estimation

In ADI, a temporal join matches two tuples that satisfy at least a predicate over vt interval fields. The Table 5.1, where i_1 and i_2 are temporal intervals, lists the predefined predicates in the platform. We can notice the following relations between the predicates:

- $Bd(I_1, I_2) = Cb(I_2, I_1)$.
- $Ed(I_1, I_2) = Ce(I_2, I_1)$
- $Inc(I_1, I_2) = Over(I_2, I_1)$
- $|Inter(I_1, I_2)| = |Bd(I_1, I_2)| + |Ed(I_1, I_2)| + |Over(I_2, I_1)| + |Inc(I_1, I_2)|$

There are two types of *temporal joins*:

- The *simple temporal join* is expressed in the form of $r_1 \bowtie_{p_{vt}(a_1.vt,a_2.vt)} r_2$ where p_{vt} is a temporal predicate, and a_1 and a_2 are respectively the join attribute of the relations r_1 and r_2. For the sake of simplification, we express a *simple temporal join* in the form of $a_1 \bowtie_{vt\_predicate(vt_1,vt_2)} a_2$
- The *composite temporal join* is expressed in the form of $r_1 \bowtie_{p_{vt}(a_1.vt,a_2.vt)AND(a_1.surID=a_2.surID)} r_2$, and involves both a temporal predicate over vt interval fields and an equality predicate over surrogate fields. For the sake of simplification, we express a *composite temporal join* in the form of $a_1 \bowtie_{vt\_predicate(vt_1,.vt_2)AND(surID_1=surID2)} a_2$

For each temporal join type, we propose an approach to estimate the result size of joining two relations r_1 and r_2.

Table 5.1: Time predicate's definitions

Temporal Predicate	Abbreviation	Definition
$Intersect(I_1, I_2)$	*Inter*	$(I_1.b < I_2.b)?I_2.b < I_1.e : I_1.b < I_2.e$
$Begin\_during(I_1, I_2)$	*Bd*	$I_2.b \leq I_1.b < I_2.e$
$End\_during(I_1, I_2)$	*Ed*	$I_2.b < I_1.e \leq I_2.e$
$Current\_at\_begin(I_1, I_2)$	*Cb*	$I_1.b \leq I_2.b < I_1.e$
$Current\_at\_end(I_1, I_2)$	*Ce*	$I_1.b < I_2.e \leq I_1.e$
$Included\_in(I_1, I_2)$	*Inc*	$I_2.b \leq I_1.b$ AND $I_1.e \leq I_2.e$
$Overlaps(I_1, I_2)$	*Over*	$I_1.b \leq I_2.b$ AND $I_2.e \leq I_1.e$

5.3.2.1 Simple Temporal Join

We adopt an approach that is based on the histograms we defined for scan estimation as well as some additional histograms.

For the sake of simplification, the subscript of histogram's names refers to join attribute. We assume that the two join attributes, let us say a_1 and a_2, have the same lifespan. This means that:

$$\{\bigcup_{i=0}^{n} t_i^1.vt \backslash t_i^1 \in a_1\} = \{\bigcup_{j=0}^{m} t_j^2.vt \backslash t_j^2 \in a_2\} \tag{5.3}$$

We also consider that all a_1 and a_2's histograms are partitioned into the same buckets. In the follows, we present the approaches for each temporal predicate.

- *Current_at_begin* ($|a_1 \bowtie_{Cb(vt_1,vt_2)} a_2|$): We assume that all a_1's tuples that intersect a bucket's interval will match a_2's tuples that start during the same bucket's interval. In consequence, we propose the formula defined in the equation 5.4. It consists in crossing all buckets that exist during the join attribute's lifespan. For each one of them, we compute the product of the number a_1's tuples that intersect that bucket's interval, using *StartBefore* and *EnfBefore*, with the value of new intervals of a_2 during the bucket's interval using the histogram H_{b2}.

$$\begin{cases} \sum_{i=1}^{b_{No}(H_{e1})} (StartBefore(instant_e) - EndBefore(instant_b)) * b_{val}(i, H_{b2}) \\ with \\ instant_b = b_{val}(i, H_{b1}) \\ instant_e = b_{val}(i, H_{e1}) \end{cases} \tag{5.4}$$

- *Current_at_End* ($|a_1 \bowtie_{Ce(vt_1,vt_2)a_2}|$): We assume that all a_1's tuples that intersect a bucket's interval will match a_2's tuples that end during the same bucket's interval. The approach to estimate $ja_1 \bowtie_{Ce(vt_1,a_2.vt)} a_2j$ is very similar $ja_1 \bowtie_{Cb(vt_1,vt_2)} a_2j$. Indeed, the estimation formula 5.5 uses the histogram H_{e2} instead of H_{b2}.

$$\begin{cases} \sum_{i=1}^{b_{No}(H_e)} (StartBefore(instant_e) - EndBefore(instant_b)) * b_{val}(i, H_{e2}) \\ with \\ instant_b = b_{val}(i, H_{b1}) \\ instant_e = b_{val}(i, H_{e1}) \end{cases} \tag{5.5}$$

- *Included_in* ($|a_1 \bowtie^T_{Inc\_in(vt_1,vt_2)a_2} j$): We assume that all a_1's tuples that are included in a bucket's interval are included in a_2's tuple intervals that intersect the same bucket's interval.

  In addition to the two histograms that we defined up to now, we define another *width-balanced histogram* H_{inc} where each of its buckets contains the number of the column's included intervals in it. In consequence, the estimation is expressed in 5.6.

$$\begin{cases} \sum_{i=1}^{b_{No}(H_{e2})} (StartBefore(instant_e) - EndBefore(instant_b)) * b_{val}(i, H_{inc2}) \\ with \\ instant_b = b_{val}(i, H_{b1}) \\ instant_e = b_{val}(i, H_{e1}) \end{cases} \tag{5.6}$$

- *Intersection*: In the case of $r_1 \bowtie_{Inter(a_1.vt,a_2.vt)} r_2$, the estimation is expressed in 5.7, based on the relation in subsection 5.3.2.

$$\begin{aligned} |r_1 \bowtie_{Inter(a_1.vt,a_2.vt)} r_2| = |r_1 \bowtie_{Bd(a_1.vt,a_2.vt)} r_2| + \\ |r_1 \bowtie_{Ed(a_1.vt,a_2.vt)} r_2| + \\ |r_1 \bowtie_{Over(a_1.vt,a_2.vt)} r_2| - \\ |r_1 \bowtie_{Inc(a_1.vt,a_2.vt)} r_2| \end{aligned} \tag{5.7}$$

5.3.2.2 Composite Temporal join

This type of joins is usually used to join an entity-column with one of its member column, i.e attribute or relationship. In consequence, there is a referential integrity constraint on the surrogate ids, i.e for each join surrogate id of the member column, there is a corresponding one in the entity-column. We also assume that a member value is always defined over of the entity instance's lifespan to which it belongs, e.g if an entity instance is defined over [14:00, 16:00[, then the member value is defined over the whole interval. Then, our estimation is:

$$|c \bowtie^{T}_{p_{vt}(c.vt,m.vt)AND(c.surID=m.surID)} m| = number\_tuples(c) * val\_per\_SurID(m) \quad (5.8)$$

c and m are respectively an entity-column and a member-column. The functions *number_Tuples* and *val_per_SurID* returns the number of entity instances and the average number of the member's values per entity instance.

5.3.3 Implementation

5.3.3.1 Statistics Generation

Computing statistics can be very costly and may heavily impact the column store insertion performances. In consequence, they are not computed as data is inserted in the columns but rather at the flush (subsection 2.3.6) because it is a completely asynchronous with data insertion. This approach implies that there are no available statistics for data that has been flushed on the disk yet. We assume that statistics of on disk stored data are enough representative.

During the flush, we generate for each column and each *SSTable* (subsection 2.3.6) the following statistics and timestamp them with the instant when the flush was launched:

- H_b and H_e to store respectively the vt intervals
- H_{inc} to store the distribution of the included intervals
- H_{surID} to store the distribution of of each surID field.

Algorithm 2 Execution plan optimization algorithm

```
1: Input:initialExecutionPlan
2: Output:optimalExecutionPlan
3: optimalExecutionPlan := initialExecutionPlan;
4: Integer i:=0;
5: while i < MAX_NUMBER_MOVES do
6:     newExecutionPlan := move(optimalExecutionPlan);
7:     if cost(newExecutionPlan) < cost(optimalExecutionPlan) then
8:         optimalExecutionPlan :=newExecutionPlan;
9:     end if
10:    i++;
11: end while
```

A scan operation may access to several SSTables which requires to combine their statistics to estimate its result size. We choose to merge the histograms and in the case of bucket overlapping, we keep the most recent ones according the query tt.

5.3.3.2 Solution Search Algorithm

We implemented a *randomized algorithm* (Algorithm 2). Considering an initial execution query plan, we randomly generate at most MAX_NUMBER_MOVES plans. At each generation, we evaluate its cost. We compare its cost the best query plan's cost and keep the cheapeast one.

5.4 Experiments

For this ongoing work, we conducted some preliminary experiments to validate its interest.

5.4.1 Query Plan

We use a simple *Select-From* query that only concerns the entity type *order* of the TPC-H/TPC-BiH benchmark, and which aims at answering the following question "Q":

> "What are the orders[1] that have been placed every day from 1/1/1992 to 1/1/1995 considering the most recent data?"

[1] we display the attributes orderkey, orderstatus, custkey, totalprice

Figure 5.4 is one possible "Q"'s execution plan. As for the previous experiments (sub-

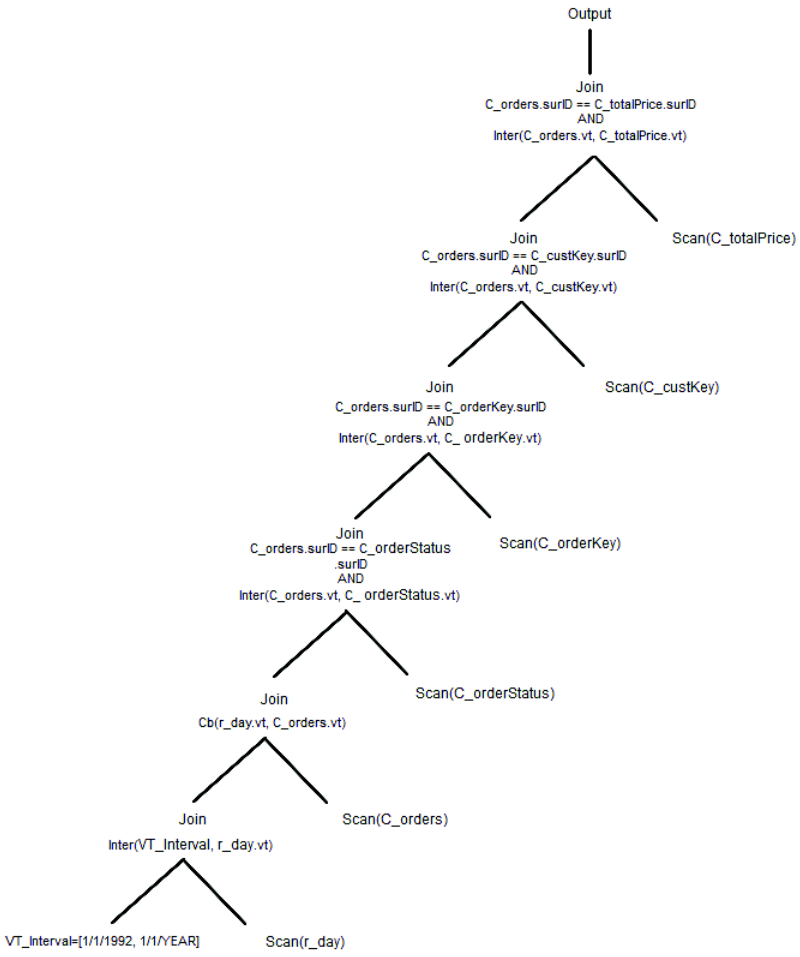


FIGURE 5.4: Q's initial execution plan

section 4.3.1), we generate a stream of events $<id,\ data,\ T>$ that consists of inserting and updating orders. The order's average lifespan is a random value from an uniform distribution over the range [20 days, 40 days]. The initial data stream contains around 340000 events (Table 5.2). It corresponds to $SF = 1$, with "SF" the scaling factor to fix the rate of the data stream.

For the sake of simplicity, the update events concern the order's vt update (40000 events) and its attribute *orderStatus* update (300000 events). This means that for a given order, its attribute *orderStatus* is updated on average 6 times.

Let us consider $YEAR = 1993$, and the result size of *Scan(C_orders)* (J1) is 100. We assume that the orders' lifespans are included within the interval $[01/01/1992, 01/01/1993[$. This means that $|J2| = 600$ since each order will match 6 values from *orderstatus*. $|J3|$

will be 600 too because each tuple from *J2* will match one tuple from *S3*. The same reasoning applies to *J4* and *J5*.

This query execution plan is not a good solution because *J2* induces a raise of the following joins' result size. Optimizing this plan using our approach consists in pushing up *J2*, so that *J2*'s previous operators will handle less data. Figure 5.5 is the optimal execution plan produced by our optimization. *J2* is pushed up to the top of the execution plan. In consequence, the previous joins's result are limited to 100 tuples.

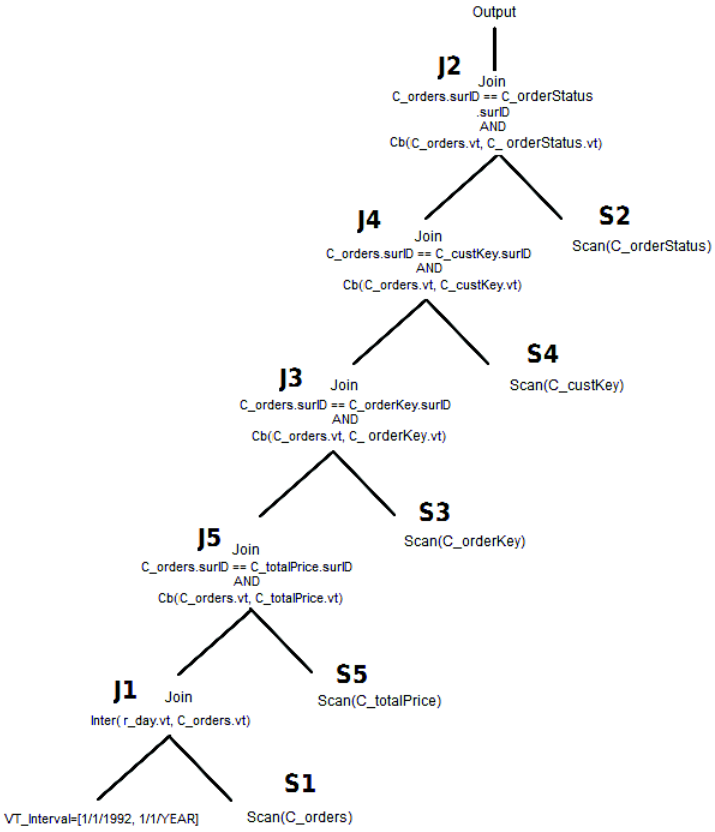


FIGURE 5.5: Q's optimal execution plan produced by our optimization

TABLE 5.2: Number of operations of the table *orders*

Relation	# of insertions	# of updates	# of deletions
Orders	50000	340000	0

5.4.2 Results

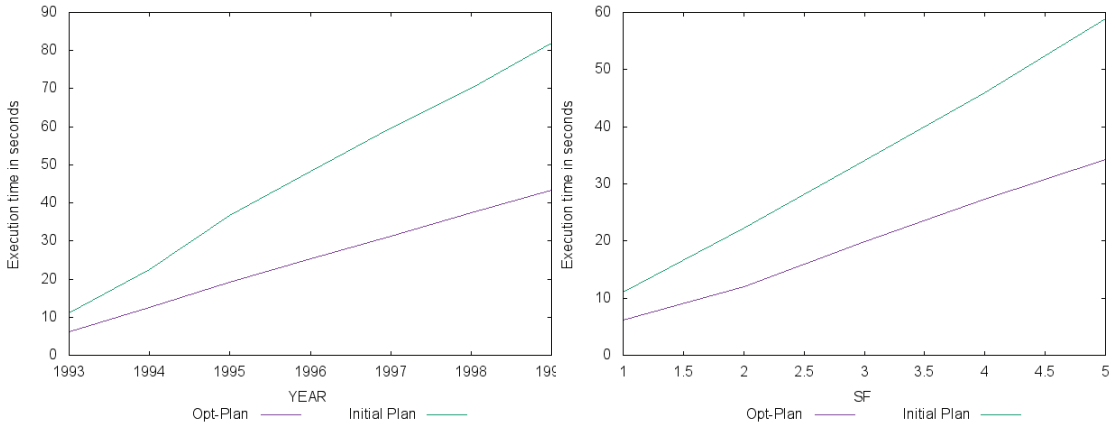
We conducted two experiments to assess the interest of our optimizer. The first one aims to evaluate the optimization gain as we vary the data size accessed by the execution plan. The second one focuses on the optimization gain as we vary a data characteristic (in our case the number of orderStatus updates).

Concerning the first experiment, we performed two tests:

- *Fixed Scalar Factor*: We inject stream for the period [1/1/1992,1/1/1996[. Then we execute "Q" considering different values of the parameter "[YEAR]" from {1993, 1994, 1995, 1996, 1997, 1998, 1999}.
- *Varying Scalar Factor*: The experimental conditions are similar to the previous test. We vary the value of *SF* from 1 to 5. For each value of *SF*, we inject the stream for the period [1/1/1992,1/1/1993[. Then we execute "Q" using the parameter [YEAR]=1993.

For both tests, we collect the plan's execution times and represent them on Figures 5.6a and 5.6b. "*Initial Plan*" is the execution plan generated by ADI based on the query, while "*Opt-Plan*" is the one generated by the optimizer based on "*Initial Plan*" and is always optimal in these experiments (Figure 5.5). "*Opt-Plan*" execution time also includes the optimization process time, i.e optimal solution search time as-well-as statistic uploading. The results show that the optimal plans require barely half of the initial plan's execution time. The optimization processing overhead is quite limited and does not reach 500 ms. Concerning the second experiment, we vary the number of the attribute *orderstatus* updates, $\#U$, while the number of order insertions is fixed to 30000. We inject data stream for the period [1/1/1992,1/1/1996[and execution the queries with *YEAR=1996*. Figure 5.7 shows the evolution of the query plan's execution time as we vary $\#U$. $\#U = 0$ means that the attribute *orderstatus* is not updated. In consequence, *J2* is equivalent to the other join operators and thus, the optimization relevent. More we increase the number of updates, more efficient the optimization is.

FIGURE 5.6: Data volume variation experiment



(A) Query execution time as function of YEAR (B) Query execution time as function of SF

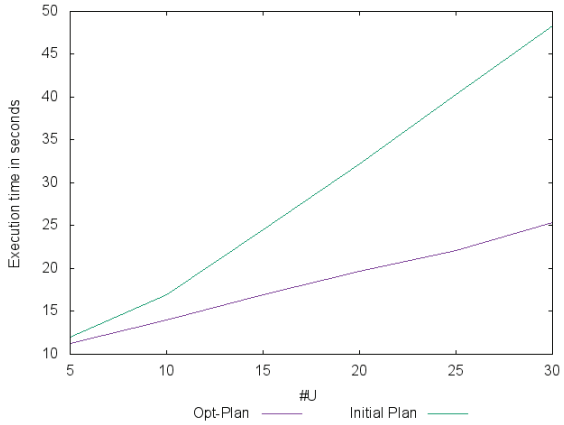


FIGURE 5.7: Query execution time as function of $\#U$

5.5 Chapter Synthesis

In this chapter, we presented ADI's first query cost-based optimizer. It focuses on the *Join Ordering* optimization which consists in ordering the execution of join operators in an optimal way. The optimizer's cost model relies on the size estimation of execution plans' operators' results. Assuming that data distribution is not dependent on the tt and that the 3 dimensions (vt, surrogate id and value) are independents, we define a set of statistics, such as histograms, on data. We choose to compute them during the data flush on disk since it does not affect ADI's data insertion performances. We adopt a *Randomized* optimization algorithm. We conducted some preliminary experiments to assess the interest of our optimizer. The results shows that the optimizer was able to divide by 2 the query execution time and the optimization time cost is acceptable.

Chapter 6

Conclusion

6.1 Summary of Contributions

Axway Decision Insight [18], an *Operational Intelligence* platform developed by Axway, enables decision-makers to make efficient operational decisions through analyzing bi-temporal data.

Since the majority of ADI's users have limited technical skills, the platform is *code-free*, i.e its use does not require any piece of code. Instead, it provides a convenient GUI that enables them to design their own applications and use them in an efficient way. This includes designing data models using ER formalism, data integration and designing queries with intuitive interfaces.

ADI's key innovation is an embedded proprietary column-oriented DBMS that has been specifically designed to meet OI requirements. It has the particularity of being natively bi-temporal, i.e it supports two temporal aspects: the *valid time* and the *transaction time* which enable to handle respectively the variation of data in the modeled reality and data update in the DB.

We presented an adaptation of the bi-temporal database benchmark, TPC-BiH [2], for OI use cases. It consists in adapting the data model as well as the database populating strategy. We used this benchmark to compare the performance of ADI with a row-oriented DBMS. The experiments showed that:

- ADI has better data insertion performances thanks, in part, to its data *append-only* strategy on disk.
- The row-oriented DBMS has better query execution performances thanks to the extensive use of indexes.

We also used this benchmark to compare several implementation designs based on data temporal characteristics [35]. The results confirms their importance during the design process on the performances.

Then we presented an optimization that is currently implemented in ADI. It consists in pre-computing all queries' aggregation operations as input data is collected and then materializing the results for future uses instead of computing them at query time [70]. This speeds up the GUI refresh and thus reduces GUI's display lag. Our contribution consisted first in aligning it with some existing approaches such as *materialized views* and second, assessing it. The experiments using our adapted version of *TPC-BiH* show that despite a computation CPU overhead, the proposed optimization improves the GUI's reactivity.

The last contribution concerns the setting up of the first ADI's query cost-based optimizer. It focus on a particular optimization, the *join ordering* [59], which consists in determining the best join order to reduce the query execution time. Up to now, the cost model focus on estimating the result's size of the query's execution plan's operators. To do this, the optimizer maintains statistics data over such as histograms to store data distribution over time. The preliminary experiments confirmed the interest of this optimization and that the optimization overhead is very limited.

6.2 Discussion and Future Works

Our current benchmark allows to assess the interest of the aggregation pre-computing optimization and its overcost, it does not assess the system's responsiveness to compute the aggregations and makes them available. Such a metric is more used in DSMS's benchmarks such as *Linear Road* [71].

Concerning the aggregation pre-computing optimization, we assume that events are usually *retroactively bounded* (subsection 4.2.3), i.e $vt_e < tt_e \leq \Delta t + vt_e$ with $\Delta t > 0$.

In practice, the value of Δ is setted by the user. A small value of Δt triggers premature computations. In consequence, the arrival of new data causes new computations and thus a CPU overcost. A bigger value reduces the re-computation risk, but raises the system latency because it waits Δt before triggering computations. One improvement would be to set automatically the value of Δt considering both latency and CPU overcost constraints, i.e determine the relationship between the valid time and the transaction of an event. We can base this improvement on *temporal dependency* works that aims to determine the relationship between events.

The cost-model optimizer is an ongoing work and many features need to be implemented to enhance its capabilities. Up to now, there are only statics on column's surrogates and valid time fields. One improvement is to define statistics, e.g histograms or dictionaries, for value fields. This will allow the optimizer to handle selection operators. The cost model that is based only on the size estimation of intermediate results generated by the execution plan operators. The enhancement of the query engine, e.g to handle distributed query executions or several implementations of a given operator, requires to enhance the cost model in order to keep its accuracy. This requires to consider additional metrics such as as transfer costs, algorithm costs, etc. More experiments need to be performed to assess the optimizer. This includes using more complex queries, a variety of data sets and comparing several optimization algorithms, see for instance [59].

Bibliography

[1] Colin White. The next generation of business intelligence: operational bi. *Information Management*, 15(5):34, 2005.

[2] Martin Kaufmann, Peter M Fischer, Norman May, Andreas Tonder, and Donald Kossmann. Tpc-bih: A benchmark for bitemporal databases. In *Performance Characterization and Benchmarking*, pages 16–31. Springer, 2014.

[3] David W McCoy. Business activity monitoring: Calm before the storm. *Gartner Research*, 2002.

[4] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[5] Hugh J Watson and Barbara H Wixom. The current state of business intelligence. *Computer*, 40(9):96–99, 2007.

[6] James Clifford and David S Warren. Formal semantics for time in databases. *ACM Transactions on Database Systems (TODS)*, 8(2):214–254, 1983.

[7] Christian S Jensen. Introduction to temporal database research. *Temporal database management*, pages 1–28, 2000. URL http://people.cs.aau.dk/~csj/Thesis/.

[8] Vassilis J Tsotras and X Sean Wang. Temporal databases. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[9] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[10] Ramez Elmasri and Shamkant B Navathe. *Fundamentals of database systems*. Pearson, 2014.

[11] Mark Levene and George Loizou. *A guided tour of relational databases and beyond*. Springer Science & Business Media, 2012.

[12] Yu Wu, Sushil Jajodia, and X Sean Wang. Temporal database bibliography update. In *Temporal Databases: research and practice*, pages 338–366. Springer, 1998.

[13] Curtis Dyreson, Fabio Grandi, Wolfgang Käfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard Thomas Snodgrass, Mike D. Soo, Abdullah Tansel, Paolo Tiberio, and Gio Wiederhold. A consensus glossary of temporal database concepts. *SIGMOD Rec.*, 23(1):52–64, March 1994. ISSN 0163-5808. doi: 10.1145/181550.181560. URL http://doi.acm.org/10.1145/181550.181560.

[14] Richard Thomas Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., 1999.

[15] Claudio Bettini, Curtis E Dyreson, William S Evans, Richard T Snodgrass, and X Sean Wang. A glossary of time granularity concepts. In *Temporal databases: Research and practice*, pages 406–413. Springer, 1998.

[16] Jae Young Lee and Ramez Elmasri. An eer-based conceptual model and query language for time-series data. *ER*, 1507:21–34, 1998.

[17] A Narasimhalu. A data model for object-oriented databases with temporal attributes and relationships. *Reporte técnico, National University of Singapore*, 1988.

[18] Azhar Ait Ouassarah, Nicolas Averseng, Xavier Fournet, Jean-Marc Petit, Romain Revol, and Vasile-Marian Scuturici. Understanding Business Trends from Data Evolution with Tornado (demo). In *Int. Conf. on Data Engineering (ICDE 2015)*. IEEE, May 2015. URL https://hal.archives-ouvertes.fr/hal-01170156.

[19] Azhar Ait Ouassarah, Jean-Marc Petit, Romain Revol, and Vasile-Marian Scuturici. Bi-temporal data modeling. Technical report, Technical report, Liris/Systar , Database Team, 2013.

[20] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[21] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[22] Shashi K Gadia. *A seamless generic extension of SQL for querying temporal data*. University of Iowa, Department of Computer Science, 1992.

[23] Leslie E McKenzie Jr. An algebraic language for query and update of temporal databases. Technical report, DTIC Document, 1988.

[24] Edwin McKenzie and Richard Snodgrass. *Supporting valid time in an historical relational algebra: Proofs and extensions*. University of Arizona, Department of Computer Science, 1991.

[25] Abdullah Uz Tansel and Lucy Garnett. *Nested historical relations*, volume 18. ACM, 1989.

[26] Christian S Jensen, Michael D Soo, and Richard T Snodgrass. Unifying temporal data models via a conceptual model. *Information Systems*, 19(7):513–547, 1994.

[27] Nicholas Rescher and Alasdair Urquhart. *Temporal logic*, volume 3. Springer Science & Business Media, 2012.

[28] Christian S Jensen and Richard T Snodgrass. Semantics of time-varying information. *Information Systems*, 21(4):311–352, 1996.

[29] Richard Snodgrass. The temporal query language tquel. *ACM Transactions on Database Systems (TODS)*, 12(2):247–298, 1987.

[30] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[31] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[32] Ramez Elmasri, Ihab El-Assal, and Vram Kouramajian. *Semantics of temporal data in an extended ER model*. Department of Computer Science, University of Houston, 1990.

[33] Manfred R Klopprogge. Term: an approach to include time dimension in the entity-relationship model. In *Proceedings of the Second International Conference*

on the Entity-Relationship Approach to Information Modeling and Analysis, pages 473–508. North-Holland Publishing Co., 1981.

[34] Heidi Gregersen. Timeerplus: a temporal eer model supporting schema changes. In *Database: Enterprise, Skills and Innovation*, pages 41–59. Springer, 2005.

[35] Christian S Jensen and Richard T Snodgrass. Temporally enhanced database design, 2000. URL http://people.cs.aau.dk/~csj/Thesis/.

[36] Martin Kaufmann, Panagiotis Vagenas, Peter M Fischer, Donald Kossmann, and Franz Färber. Comprehensive and interactive temporal query processing with sap hana. *Proceedings of the VLDB Endowment*, 6(12):1210–1213, 2013.

[37] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1173–1184. ACM, 2013.

[38] Christian S Jensen, Leo Mark, Nick Roussopoulos, and Timos Sellis. Using caching, cache indexing and differential techniques to efficiently support transaction time. 1990.

[39] Christian S Jensen, Leo Mark, and Nick Roussopoulos. Incremental implementation model for relational databases with transaction time. *Knowledge and Data Engineering, IEEE Transactions on*, 3(4):461–473, 1991.

[40] Richard T Snodgrass. Temporal databases. In *Theories and methods of spatio-temporal reasoning in geographic space*, pages 22–64. Springer, 1992.

[41] Christian S Jensen and Richard Snodgrass. Temporal specialization and generalization. *Knowledge and Data Engineering, IEEE Transactions on*, 6(6):954–974, 1994.

[42] Mario A Nascimento and Margaret H Eich. Decision time in temporal databases. In *Proceedings of the Second International Workshop on Temporal Representation and Reasoning*, pages 157–162. Citeseer, 1995.

[43] S Chakravarthy and SK Kim. Semantics of time-varying information and resolution of time concepts in temporal databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases, Arlington, TX*, 1993.

[44] Sharma Chakravarthy and Seung-Kyum Kim. Resolution of time concepts in temporal databases. *Information Sciences*, 80(1):91–125, 1994.

[45] Seung-Kyum Kim and Sharma Chakravarthy. Modeling time: Adequacy of three distinct time concepts for temporal databases. In *Entity-Relationship Approach—ER'93*, pages 475–491. Springer, 1994.

[46] Christian S Jensen. *A consensus test suite of temporal database queries*. University of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science, 1993.

[47] Margaret H Dunham, Ramez Elmasri, Mario A Nascimento, and Marion Sobol. *Benchmarking temporal databases: A research agenda*. Citeseer, 1995.

[48] Paul Werstein. A performance benchmark for spatiotemporal databases. In *In: Proc. of the 10th Annual Colloquium of the Spatial Information Research Centre*, pages 365–373, 1998.

[49] Mohammed Al-Kateb, Alain Crolotte, Ahmad Ghazal, and Linda Rose. Adding a temporal dimension to the tpc-h benchmark. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 51–59. Springer, 2013.

[50] RT Snodgrass, MH Böhlen, CS Jensen, and A Steiner. Adding valid time to sql/temporal. ansi x3h2-96-151r1, iso–ansi sql/temporal change proposal, iso. Technical report, IEC JTC1/SC21/WG3 DBL MCI-142, 1996.

[51] Richard T Snodgrass, Michael H Böhlen, Christian S Jensen, and Andreas Steiner. Adding transaction time to sql/temporal. *ISO-ANSI SQL/Temporal Change Proposal, ANSI X3H2-96-152r ISO/IEC JTC1/SC21/WG3 DBL*, 1101:143, 1996.

[52] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql: 2011. *ACM SIGMOD Record*, 41(3):34–43, 2012.

[53] Daniel Barbar'a, William DuMouchel, Christos Faloutsos, Peter J Haas, Joseph M Hellerstein, Yannis Ioannidis, HV Jagadish, Theodore Johnson, Raymond Ng,

Viswanath Poosala, et al. The new jersey data reduction report. In *IEEE Data Engineering Bulletin*. Citeseer, 1997.

[54] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. 2002.

[55] Jürgen Krämer and Bernhard Seeger. *A temporal foundation for continuous queries over data streams*. Univ., 2004.

[56] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[57] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320. VLDB Endowment, 2003.

[58] Sirish Chandrasekaran and Michael Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 348–359. VLDB Endowment, 2004.

[59] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal—The International Journal on Very Large Data Bases*, 6(3):191–208, 1997.

[60] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[61] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In *ACM SIGMOD Record*, volume 25, pages 35–46. ACM, 1996.

[62] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. Citeseer.

[63] Arie Segev, Himawan Gunadhi, Rakesh Chandra, and J George Shanthikumar. Selectivity estimation of temporal data manipulations. *Information Sciences*, 74 (1):111–149, 1993.

[64] Giedrius Slivinskas, Christian S Jensen, and Richard Thomas Snodgrass. Adaptable query optimization and evaluation in temporal middleware. In *ACM SIGMOD Record*, volume 30, pages 127–138. ACM, 2001.

[65] Rosana SG Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, volume 93, pages 493–504, 1993.

[66] Arun N Swami and Balakrishna R Iyer. A polynomial time algorithm for optimizing join queries. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 345–354. IEEE, 1993.

[67] Arun Swami and Anoop Gupta. *Optimization of large join queries*, volume 17. ACM, 1988.

[68] Arun Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *ACM SIGMOD Record*, volume 18, pages 367–376. ACM, 1989.

[69] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989, 1989.

[70] Azhar Ait Ouassarah, Nicolas Averseng, Xavier Fournet, Jean-Marc Petit, Romain Revol, and Vasile-Marian Scuturici. Bi-temporal Query Optimization Techniques in Decision Insight. In *(BDA 2015)*, 2015. URL https://hal.archives-ouvertes.fr/hal-01170156.

[71] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.