Relational Support for Flexible Schema Scenarios

Srini Acharya
srinia@microsoft.com

Peter Carlin
peterca@microsoft.com

Cesar Galindo-Legaria
cesarg@microsoft.com

Krzysztof Kozielczyk[*]
krkozie@microsoft.com

Pawel Terlecki
pawelt@microsoft.com

Peter Zabback
pzabback@microsoft.com

Microsoft SQL Server
One Microsoft Way,
Redmond, WA 98052

ABSTRACT

Efficient support for applications that deal with data heterogeneity, hierarchies and schema evolution is an important challenge for relational engines. In this paper we show how this flexibility can be handled in Microsoft SQL Server. For this purpose, the engine has been equipped in an integrated package of relational extensions. The package includes sparse storage, column set operations, filtered indices, filtered statistics and hierarchy querying with OrdPath labeling. In addition, economical loading of metadata allow us to answer queries independently of the number of columns in a table and drastically improve scaling capabilities. The design of a prototypical content and collaboration application based on a wide table is described, along with experiments validating its performance.

1. INTRODUCTION

There is a fast growing market of large-scale content and collaboration applications and messaging systems (e.g. Microsoft Exchange). Those systems support thousands of concurrent users, each defining document repositories with widely varying schemas. In addition, queries against these systems need to return results interactively and consistently in just a few seconds.

These applications will benefit from data models that go beyond the traditional schemas in relational systems. While normalized schemas are rather static and do not contain a large number of properties, large scale content and collaboration applications deal with heterogeneous collections. It is not uncommon for these applications to consider tens of thousands of different properties, where only a small fraction pertains to a particular entity. Storing such collections is an important challenge for product catalogs, document management, knowledge extraction, XML indexing ([8]) or object-oriented stores. Moreover, these applications usually require hierarchies, multi-valued properties and frequent schema changes. We use the term *flexible schema* to reflect such extensibility and heterogeneity of a data model [17].

This paper presents a set of features built into Microsoft SQL Server 2008 to explicitly support flexible schemas implemented as sparse wide tables with current limits of up to 300k columns, 1k indexes, and 300k distribution statistics. It describes the relevant design changes to the architecture of the server through system catalogs, query compilation and storage data structures. A prototypical content and collaboration application, similar to Microsoft Office Sharepoint Server, is used for illustration purposes.

The classic relational approaches of a property bag or ISA relation decomposition fail to handle heterogeneous collections efficiently ([3, 17]). For example, storing arbitrary objects in relational databases usually results in complex multi-table schemas and requires relevant translation instructions in object-relational mappers. This fact leaves application developers in an unpleasant quandary. One may consider specialized solutions, but they deprive him from a well-developed services ecosystem including data management, security, reporting, and application development tools. Also, additional integration costs should be taken into account, as significant components of business systems are most often implemented in a relational manner and receive expected support. A common surface of SQL seems to be broad enough to express heterogeneity and make specialized optimizations transparent to users. In fact, [6, 3] give solid insights that the problem can be efficiently handled by means of wide tables as long as they are accompanied with appropriate storage, transportation and indexation approaches.

In [17], two categories of flexible schemas are distinguished: unclustered and clustered. While the first one refers to strongly heterogeneous entities that do not form any classes in terms of attributes they are characterized with, for the second one, such classification is possible, e.g. documents, contacts or pictures. Moreover, if classes can be arranged in a hierarchy like product categories, object types etc., one deals with a hierarchical case. Otherwise, we say it is a flat case. The hierarchical case is attractive due to its generality and interesting usage scenarios.

A flexible schema implemented as a wide table inevitably leads to sparseness and a large number of columns. The

[*]Work done when the author was a PhD student at University of Wroclaw, Poland.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

package we propose address challenges in terms of efficient storage, processing, model modification and scaling. It is important to note that these features do not extend the relational model itself but rather augment its implementation. This ensures the full utilization of existing system services, including query processing and optimization.

In our proposition efficient querying of hierarchical data is addressed with the OrdPath labeling ([13]), which provides a compact scalar representation of position in a hierarchy tree. Sparse wide tables are stored by means of interpreted storage ([3]), which achieves significant savings in space as well as access time. In order to enable the efficient retrieval and manipulation of a large numbers of columns, we introduced a logical scalar representation that groups sparse columns in a *column set*. Economical data indexation in terms of space and DML is addressed with filtered indices ([17]). Besides mechanisms used for efficient query execution, adequate selection and query optimization support is crucial. In this area, we propose the idea of filtered statistics, which remain complementary to our approach for indexation. The proposition is completed with several optimizations that ensure scaling in terms of columns and statistics on a single table.

The presence of different classes in a clustered flexible schema implies an underlying data organization in a wide table. In fact, instances belonging to the same classes together with associated properties form logical subtables. It should be emphasized that a sparse column set, filtered indices and statistics allow us to support the subtables the same way regular tables are supported.

The package itself represents a value as an integrated solution for flexible schemas. Among its elements, four of them contribute to the field: new data organization in interpreted storage, sparse column set in data transportation and manipulation, filtered statistics in cardinality estimation and techniques that ensure scalability of wide tables in terms of columns.

The paper is organized as follows. Section 2 outlines a prototypical flexible schema database application. This application serves as running example throughout the rest of the paper. In Sect. 3 a labeling scheme for hierarchical data is described. Sparse storage and operations with a sparse column set are discussed in Sect. 4 and 5.Support for indexation and cardinality estimation is covered in Sect. 6 and 7. Large scale capabilities are given in Sect. 8. Section 9 outlines previous work. In Sect. 10 experimental results for individual features and the integrated package are provided. We finally summarize our contribution in Sect. 11.

2. DETAILED EXAMPLE

There is a number of real life applications like document management systems, product catalogs or location management systems that fit the pattern of a flexible schema. In this section we describe a simplified version of a prototypical content and collaboration application that is similar to Microsoft Office Sharepoint Server, and discuss benefits of the presented design. This example is also utilized later to describe the feature package we implemented and perform experimental validation.

2.1 Content and Collaboration Application

A content and collaboration system is an important work platform in modern corporations. Users can conveniently store and manage various content in a common environment

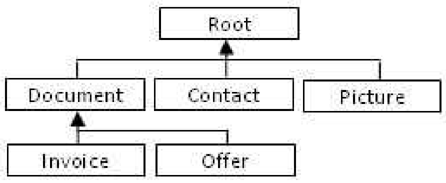


Figure 1: A fragment of the type hierarchy in our sample content and collaboration application

of integrated tools. Such an application not only allows storage of heterogeneous content, like documents, pictures or contact information, characterized by a fixed set of type specific properties, but also provides users with the flexibility to annotate their content with custom properties. The content types held in the system belong to a common hierarchy. A simplified version of the hierarchy in our sample application is presented in Fig. 1.

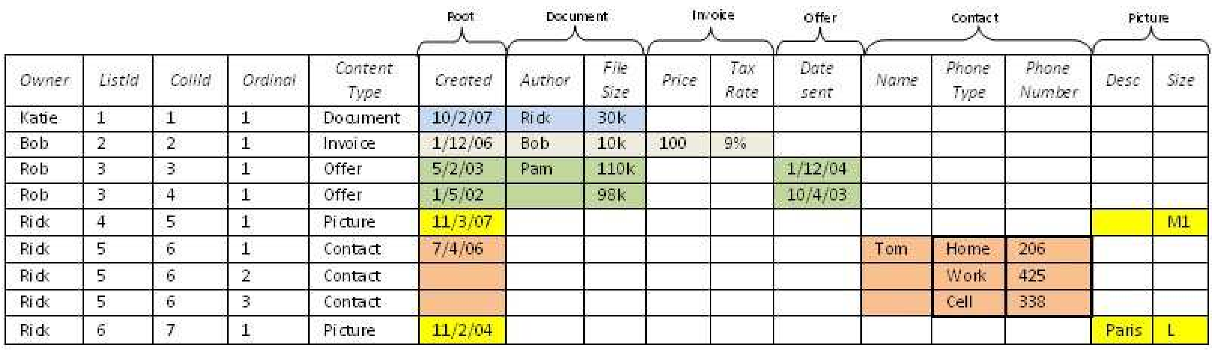
Users organize their data in lists of elements of a particular content type. Each content type is associated with a specific set of properties that is inherited and extended in subtypes. At the top of the type hierarchy there is a generic root that defines common properties, usually of administrative character, e.g. *Date created.* Content types become more and more specialized when one goes down the hierarchy. Let us look closer at the *Document* type in our example. Besides characteristics inherited from *Root*, it may also introduce properties like *Author*, *Reviewer* that are shared by its subtypes: *Offer*, *Invoice* etc. Accordingly, the definition of *Offer* can be extended with the properties *Date sent*, *Total value* etc. Depending on a particular system a type hierarchy can be both arbitrarily wide and deep. In addition to the type specific properties, our sample schema also models flexibility for a user to extend predefined property sets with custom annotations and better suit personal needs. This capability can be seen as adding new content types and impact the hierarchy's extensibility.

Typical retrieval access patterns in the system include displaying the content lists owned by a user, e.g. lists in the user's home page, or accessing all content of a specific type, e.g. all invoices for the last month. New lists are created with a common content type for its future items. Each item that is added can be described with properties defined for this content type. Another use case is an update of a specific property to a selected number of items in a user's list, e.g. *DateTaken* for pictures stored in a user's pictures list. As far as administrative operations are concerned, a large volume of data may be retrieved and copied to a backup table.

2.2 Flexible Schema Design

The main characteristics of such a content management system is heterogeneous collection of instances of content types from a certain hierarchy.

Both extensibility and efficiency of such applications can be achieved by storing entities of a heterogeneous types in a single wide table. Then, for each content type, there is a logical subtable within the whole table that consists of rows and columns associated with this content type. A sample table, referred to as *ListItem*, is given in Fig. 2. It contains lists of items for users of the system. For example, *Katie*,



					Root	Document		Invoice		Offer	Contact			Picture	
Owner	*ListId*	*CollId*	*Ordinal*	*Content Type*	*Created*	*Author*	*File Size*	*Price*	*Tax Rate*	*Date sent*	*Name*	*Phone Type*	*Phone Number*	*Desc*	*Size*
Katie	1	1	1	Document	10/2/07	Rick	30k								
Bob	2	2	1	Invoice	1/12/06	Bob	10k	100	9%						
Rob	3	3	1	Offer	5/2/03	Pam	110k			1/12/04					
Rob	3	4	1	Offer	1/5/02		98k			10/4/03					
Rick	4	5	1	Picture	11/3/07										M1
Rick	5	6	1	Contact	7/4/06						Tom	Home	206		
Rick	5	6	2	Contact								Work	425		
Rick	5	6	3	Contact								Cell	338		
Rick	6	7	1	Picture	11/2/04									Paris	L

Figure 2: Sample flexible schema for a prototypical content and collaboration application, brackets indicate properties introduced by each content type and bolded frames distinguish multi-valued properties

Rob and *Rick* own 1, 2, 3 different lists, respectively. The approach adopts the model of having a single column per each unique property defined in the content type hierarchy (Fig. 1). The columns that do not pertain to the content type of a particular row are equal to null and the whole table remains sparse ([3, 17]).

In some cases, properties may be multi-valued, like *Phone type* and *Phone number*. For example, in object-oriented frameworks they are usually treated as a part of an entity definition and organized in dependent collections. Also here, wide tables provide a convenient way to handle ordered and unordered value collections for single properties. Namely, each entity is specified by multiple rows with the same value of *CollId*. If the order is meaningful, an additional column *Ordinal* is employed. Note that the same query can be used to retrieve entities with simple and with multi-valued properties. Also, an additional overhead in space originating from common administrative attributes is insignificant and can be reduced by prefix column compression. In our example, in the class *Contact*, there are two multi-valued properties *Phone type* and *Phone number*, whose values are ordered collections. In this case, both properties are tightly-coupled and form a complex type, but this is not a requirement. A sample entity with 3 different phones is encoded as the collection with *CollId=6* belonging to the list with *ListId=5.*

2.3 Metadata vs. Data

When an application uses subtables in the way we have described, it is effectively shifting traditional "metadata content" into the "data" itself. Subtables are not designated in queries by a pre-declared table name, but rather by predicates on a discriminator column. Bypassing relational metadata declaration simplifies application development and administration, especially when the number of subtables is expected to be dynamic, or when they are hierarchical. For example, queries that operate at different levels in a hierarchy can be written in a straightforward way on a single table, instead of requiring a (potentially variable) number of unions and joins.

This shift from "metadata" into "data" provides flexibility, but it comes at a price, which can be examined in light of how relational implementations exploit metadata. In terms of query processing, table metadata is the traditional anchor for indices and statistics, which are made available to the optimizer as soon as parsing is complete.

Consider a query written against pre-declared tables, and compare it with an equivalent that uses subtables instead. To reach comparable query behavior, one first needs to enable the creation of indices and statistics for subtables, then extend query optimization to identify and utilize applicable metadata. In particular, identifying relevant metadata for a query changes from simple name lookup (table name in the query) to operations based on predicate inferencing (predicates on discriminator columns).

3. QUERYING HIERARCHIES

In this paper we consider a general case of a flexible schema with a hierarchy. If the implementation is based on wide tables, one can observe a hidden schema consisting of logical subtables. Each of them refers to a particular class in the hierarchy, a content type in our prototype.

Any representation of class membership should ensure two important goals: efficient data retrieval and cheap modifications of a schema. Note that, due to differences in class definition, one may expect that the majority of queries and DML operations will address separate classes. Let us look closer at an efficient implementation of two common cases presented in [17]. A flat scenario, i.e. without any hierarchy of subtables, can be naturally implemented by a single numeric discriminating column that stores a class number for each row. Each property refers to only one logical subtable, thus, a desired range of rows can be retrieved by means of a simple equality predicate. Also, values used for encoding have strictly nominative interpretation, which does not require any renumbering when changes to the logical schema are made. On the other hand, the case of a hierarchy needs a more subtle discriminator, since each property belongs to the class it is introduced with, and to all its descendants. A prefix addressing allows us to retrieve rows from a particular class by means of a range predicate. However, possible discriminating values are ordered and of variable length, which makes both efficient querying and extensibility a challenge.

To address the aforementioned problems, the analogy between a hierarchy in a flexible schema and a hierarchical structure of XML documents has been exploited. SQL Server uses the OrdPath labeling ([13]), originally designed to deal with a general case of schemaless documents. Each class has a label associated with it. This label is defined as a sequence of ordinals separated by dots, like 1.2, 4, 2. The multi-component labels may be introduced due to careting-

Table 1: Encoding of classes in our sample hierarchy

Node	Path	Encoding (L_i, O_i, F_i)
Root	1	01 001 1
Document	1/1	01 001 1 01 001 1
Invoice	1/1/1	01 001 1 01 001 1 01 001 1
Offer	1/1/2	01 001 1 01 001 1 01 010 1
Contact	1/2	01 001 1 01 010 1
Picture	1/3	01 001 1 01 011 1

in, covered further. The address of a given subclass is a concatenation of labels on the path from the root to this class, where each two labels are separated by a slash, e.g. 1.2/3/1, 2/3/1.

In OrdPath a class address is represented in a compressed format by a sequence of triples $(L_i, O_i, F_i)_{i \in \{0,..,n-1\}}$. The first two components encode an i-th ordinal. They are bitstrings of variable-length and L_i specifies the length of O_i in bits. F_i is a single bit equal to 0, when the ordinal is followed by a dot and to 1, otherwise. A prefix-free encoding is used for the lengths L_i, which guarantees that their ends are always recognizable. It also preserves the original ordering of addresses and allows us to perform more efficient comparisons of compressed sequences.

Besides performance of data retrieval, the scheme offers a significant support for extensibility. In fact, most often there is no meaningful order between sibling classes, therefore, a new subclass may obtain the next ordinal for the current maximum. In case of a deletion one simply invalidates a given class label. An alternative is to track these ordinals and reuse them later. Since, in our scenario, these operations are relatively rare, the first option is more applicable. Interestingly, when one considers an ordered hierarchy tree, OrdPath provides efficient insertion routines for arbitrary positions. In particular, adding new classes at the left end of all subclasses of a given class is implemented by using negative values and insertion between any two existing subclasses is performed by careting in ([13]). On the whole, in both scenarios of unordered and ordered hierarchy trees, any renumbering of existing classes is avoided, as well as, manipulations in the associated wide table.

Addresses in a hierarchy are introduced as a new SQL type, HierarchyId, and deployed as a respective CLR system type. Values are internally treated as binary columns. The fact that the logical ordering of addresses is preserved in the physical representation is crucial for building indices and statistics on such columns. To support the considered scenarios, we defined a specific operation that is internally rewritten before relational processing takes place. The fact that a column *H1* is a descendant of *H2* is expressed by *H1.IsDescendant(H2)* and after translation becomes the range predicate:

$$\text{H2} >= \text{H1 and H2} <= \text{H1.DescendantLimit()}.$$

The method *DescendantLimit* returns a binary string that is greater than any child of the node $H1$ and smaller of any its sibling H, for which $H > H1$. For the sake of brevity, the encoding of a particular *class* is handled by the notation *encoding*(*class*).

In our example, all rows belonging to the content type *Offer* can be obtained by the following query:

```
SELECT *
FROM ListItem
WHERE ContentType.IsDescendant(encoding('Offer'))
```

Table 2: A sparse vector format

Header	Column buckets	Column ids	Value position	Values

4. SPARSE STORAGE

A large number of columns and sparseness are two intrinsic properties of flexible schemas. Indeed, if we deal with the clustered case, a wide table groups heterogeneous entities that share common characteristics in logical subtables. In consequence, each subtable introduces new properties and increases a total number of columns. Furthermore, since each subschema makes use of a fraction of all defined columns, the whole table becomes sparse. Similarly, unclustered schemas have extensible property spaces, while only few properties pertain to each tuple.

As argued in [3], sparse wide-tables require an efficient storage mechanism that would handle a huge overhead of meaningless null values and, thus, ensure scalability. In fact, the majority of contemporary databases use a classical positional storage. It assumes that values are held always on the same positions resulting from a fixed order and size in system catalogs. It guarantees fast access to each value, however, it also uses all assigned space to represent each empty value. The modification proposed in PostgreSQL maintains a bitmap of null values in each row and saves on additional space when a column is empty. This benefit is traded off more expensive computation of a value position. Although this is much more economical, both approaches use non-zero space to store null values and do not scale for large number of columns. This problem can be overcome by using an interpreted storage. We indicate major differences between a sparse data format implemented in SQL Server and the approach proposed in [3].

Users may need to store data of different characteristics in a single table. Therefore, interpreted storage is enabled together with positional storage for fixed- and variable-size data. A column can be marked as sparse, which implies that it belongs to the interpreted part. This part, referred here to as a sparse vector, is stored as a complex variable-size column of the format sketched in Tab. 2.

As it was discussed, in wide tables, only few columns are present in each row. Note that they can be represented by column-value pairs, while the rest is implicitly equal to null. In [3], the pairs are held in a list. This approach remains efficient when the actual number of populated columns is small. However, tuples that belong to classes at lower levels of a hierarchy may suffer from linear access complexity. We address this issue with two simple optimizations. First of all, it is very likely that columns introduced with a type were added together and their identifiers have close values. For fast negative response, the sparse vector contains buckets that refer to identifier ranges. Based on the ranges one can quickly check if a column is populated or not. Otherwise, the sorted column identifiers are searched with the binary search algorithm. If found, a respective position associated with the column id is used to retrieve a value from the data

Table 3: A result set for "SELECT *" in our example

ContentT	Coll	Ord	Created	Own	SparseColumnSet
Document	0	0	10/2/07	Rick	$<Author> Rick </Author> <FileSize> 30k </FileSize>$
Picture	0	0	11/3/07	Katie	$<Size> M1 </Size>$

part.

5. SPARSE COLUMN SET

Even if a large number of sparse columns is stored efficiently, processing may still remain a challenge. Since each sparse column is treated as a regular column in query processing and transport, as long as a user knows which columns to specify, a wide table can be easily processed by referencing these columns. However, when one deals with unstructured data or operates on data from multiple logical subtables at the same time, it is hard to decide which columns are necessary. Also, inevitably, only some of them are populated for each row.

Given such a request of heterogeneous data, let us consider a traditional way of retrieving all available columns with a "SELECT *" query. As it was pointed out, the ones that are actually present may vary from row to row causing an overhead in two ways. First, a result set is sparse. If we still provided every sparse column as other regular columns, a client would end up obtaining all columns defined for every row, even though they are not interesting. Moreover, many of them may not be even populated in the result set at all. Second, uploaded column metadata can be extremely noticeable for OLTP workloads, when a few rows are retrieved at a time.

Instead of proposing an efficient strategy for sending a format of a wide result set, SQL Server implements the idea of a *sparse column set*, which makes a physical sparse vector be accessible through a special computed column of a schemaless XML type. Sparse vector is seen as a flat document, where names of populated columns are used as tags to store respective values. A sparse column set can be referred explicitly by a user and is implicitly returned with all regular columns for the "*" case. Note that processing on a client side becomes also more efficient. The XML document can be easily interpreted to get values of underlying columns, instead of looping over a possibly large number of sparse columns.

Let us assume that the type hierarchy in our example is wide and all columns introduced at lower levels are marked as sparse. For a request of all items created after $'1/1/07'$, one obtains the result set in Tab.3.

Data manipulation is another powerful application for a sparse column set. In case of bulk operations, it is inconvenient and inefficient to specify all columns explicitly. In fact, only the columns meaningful for each row should be processed. To achieve this goal, a sparse column set is treated as updatable. Let us consider the following template that allows us to populate any logical subtable by providing values of sparse columns with a sparse column set:

```
INSERT INTO T(ContentType,SparseColumnSet)
VALUES ('Picture',
' <Desc>Fall</Desc><Size>M2</Size>')
```

Last but not least, heterogeneous definitions of logical subtables result in statements of different patterns, when columns are specified explicitly. This fact is particularly adverse in terms of parameterization and caching. Also here, common templates with a sparse column set ensure compile time savings.

6. DATA INDEXATION

Although heterogeneous collections are represented by a single wide table, queries are issued against logical subtables. Therefore, the system is expected to provide efficient data retrieval structures at this implicit level.

While designing a proper indexation solution for the flexible schema scenario, one encounters two evident challenges. First of all, new secondary structures cannot add significant storage overhead on the base table. Second, when a large number of columns is indexed, data manipulation becomes expensive. Note that, if one considered classic indices for this purpose, either storage or DML execution time would increase linearly with the number of defined indices ([17]). Given a scale and time-critical requirements of dynamic OLTP applications, this solution would not be acceptable.

Both problems stated above are a direct consequence of natural sparseness of wide-tables. As it was pointed out, only a fraction of null values that belong to logical subtables is meaningful and, thus, interesting for retrieval, whereas the vast majority remains not applicable. One of the solutions is to use a sparse index ([6]) that takes into account only non-null values. It is highly economical in terms of storage and, in case of an update of a row, only indices including affected rows are maintained. A noticeable drawback is that a sparse index cannot be treated as an equivalent of a regular index for a logical subtable. In particular, it does not cover rows with meaningful null values, which limits its usability, e.g. back-joins, seeking missing values, etc. To overcome these issues, we incorporated the more general approach of partial indexation ([18]), that is represented in our framework by a filtered index. Syntactically, the feature is surfaced by allowing a WHERE clause in the index definition, as follows:

```
CREATE INDEX inx ON table(indexed columns)
WHERE p(predicate columns)
```

The WHERE clause is orthogonal to other index qualifiers such as partitioning, included columns or fill factor. The predicate p specifies a subset of rows that are to be stored in the index.

In the remainder of this section, we present an overview on the feature, more details on implementation and use cases can be found in our previous work ([17]).

6.1 Querying

Regular indices support data retrieval in several ways, like efficient scans, seeks or providing tuples in a desirable order. The same properties hold for filtered indices as well, however, their application is restricted to a subset of a base table specified by a predicate.

In the clustered scenario, one uses a discriminating column to distinguish particular classes. As it was explained above, it is sufficient to use equality conditions to indicate class membership in flat collections, whereas, for hierarchies, it is appropriate to use range predicates that address a given inheritance subtree. Coming back to our sample schema, let us create a filtered index on the column *DateSent* for rows in the class *Offer*.

```
CREATE INDEX inx ON ListItem(DateSent)
WHERE ContentType.IsDescendant(encoding('Offer'))
```

Every query that requests tuples from this range can potentially make use of this index. Precisely, an index can be used when its predicate subsumes (matches) the query's predicate, i.e. the index has entries for all the rows required to answer the query. Logic inferences are performed by the matching engine component ([17]). As soon as the subsumption is proved the index is treated as any other index in optimization. The whole logic is executed only when index candidates are considered, thus, the presence of a predicate has almost no effect on plan compilation. At the same time, it is often the case that a query requests exactly the range of rows referenced in an index and applies an additional residual predicate, which allows us to scan the whole index and simplify the whole predicate.

6.2 Maintenance

Data manipulation changes a content of a base table and, in consequence, it requires maintenance of all dependant secondary structures. In particular, a regular index on changed columns has to be updated regardless of the rows that are affected. However, in case of partial indexation, there are opportunities for significant savings.

In SQL Server, in order to process DML statements like regular queries, the changes to be applied to a base table are modeled by a set of rows called a delta stream ([10]). As far as execution is concerned, the number of possible actions that can be performed on an index for each tuple is different from a classic case. Namely, an entry is inserted (deleted) only if new (original) values satisfy the index's predicate. The case of updates is more complex, since one may need to insert, delete, update or ignore a row depending on if a predicate is met for old and new values (see [17] for details). This logic is implemented by preceding a relevant update operator with a filter, which selects rows satisfying the index's predicate.

During plan generation it is often the case that a delta stream is characterized by some constraint, e.g. explicit assignment list or predicates. This information may be used to infer that the predicate of a given index is contradictory or tautological and simplify maintenance. Note that, if it can be proved that none of rows of a delta stream affects an index, we skip it in maintenance. Similarly, one may omit putting a filter before an update operator for an index that is guaranteed to be always affected. The necessary inferences are also made by means of the matching engine.

On the whole, maintenance cost for a filtered index as compared to a regular one is much lower due to fewer data accesses. This effect is even stronger if the index's predicate is selective in a considered delta stream. However, additional overhead is introduced in terms of CPU due to evaluation of a filtering expression. In certain cases it is possible to eliminate this overhead and at the same time greatly reduce the size of the execution plan. It is worth to emphasize that, although general, the approach is especially efficient for implementing flexible schemas. First of all, predicates are very simple, thus, not expensive to evaluate. Second, regular updates address contents of logical subtables and do not change the discriminating column. In consequence, filtered indices logically not related to a target subtable are eliminated from a plan thanks to contradictions. Also, indices that pertain only to this subtable are treated as regular indices and a maintenance plan does not have to be instrumented with additional filters.

7. FILTERED STATISTICS

7.1 General Idea

Any state of the art database engine maintains statistical information describing the data distribution of individual columns or set of columns in a table. These optimizer (or distribution) statistics are crucial for cardinality estimation and selection of efficient execution plans. Distribution statistics are on entire tables. For example, statistical information on the *Created* column describes the distribution of the creation time stamp across all rows stored in the *ListItem* table. As discussed above, queries on heterogeneous collections stored in wide tables are in fact queries against logical subtables. Therefore, distribution information on all rows in a wide table is inadequate. Since queries are against logical subtables, it is desirable to maintain distribution statistics on these subtables. This follows the justification used for filtered indices.

Due to space limitations, we provide only an overview on the feature and emphasize its role for a flexible schema implementation.

7.2 Statistics Creation

Analogously to the creation of filtered indexes we extended the syntax for statistics creation allowing a WHERE clause to specify the subset of rows we want to define statistics on:

```
CREATE STATISTICS s ON table(column1, column2, ...)
WHERE p(predicate columns)
```

As with filtered indexes, the WHERE clause is orthogonal to other statistics specifiers (e.g. sample rate). Columns referenced in the predicate need not to be a subset of the columns we create statistics on; but they need to be from the same table, which implies that filtered statistics are single table statistics.

Internally, an unfiltered create statistics statement is implemented by a (potentially sampled) scan of the underlying table. Filtered statistics are created using the same scanning/sampling infrastructure. Before a row is considered for the filtered statistics, the filter predicate is evaluated and only qualifying rows are used to calculate the data distribution. If there is a (potentially filtered) index that provides required columns as well as subsumes a filter predicate, it will be used to reduce the I/O overhead for the statistics query.

As with regular (unfiltered) indexes, SQL Server will create statistics as a side effect of an index create. Consequently, the creation of a filtered index, implies the creation full scan filtered statistics with the same filter expression.

7.3 Use in Cardinality Estimation

To understand how the optimizer exploits filtered statistics, we give a very brief introduction on the usage of unfiltered statistics. Any table column referenced in a query predicate or group by expression is considered interesting for statistics loading. If statistics do not exist on a column in question and the auto creation of statistics is enabled, SQL Server will create these statistics on the fly. Let us consider a predicate on two columns of the *ListItem* table (Fig. 2):

$Author = 'Pam'$ *and*
ContentType.IsDescendant(encoding('Offer')).

The system will load, after potentially auto-creating, statistics on *Author* and the OrdPath encoding of *ContentType*. After estimating the selectivity of the partial expression $Author = 'Pam'$, this selectivity is applied to the statistics on the OrdPath encoding of *ContentType* and vice versa. To do this, we have to assume statistical independence of *Author* and *ContentType*. We load the full table statistics on the *Author* column, estimate how many of the rows qualify for $Author = 'Pam'$ and then apply the estimated selectivity of the *ContentType* predicate to this estimate.

If we have filtered statistics on *Author* with the filter predicate *ContentType.IsDescendant(encoding('Author'))*, there is no need to apply the estimated selectivity of this predicate to the statistics on *Author*. Whenever we load statistics on a column we consider the query predicate and load filtered statistics for which the filter predicate subsumes the query predicate. In other words, the set of rows defined by the query predicate is a subset of rows defined by the statistics filter predicate. In case of multiple matching filtered statistics, we choose the one defining the smallest subsuming set.

Importantly, in case the query predicate specifies a true subset of the rows specified by a subsuming filter predicate, we still need to apply the partial (or residual) predicate. For example, let us consider the query predicate *ContentType.IsDescendant(encoding('Offer'))* and use statistics valid for rows in two sibling classes with the filter predicate:

(ContentType.IsDescendant(encoding('Offer')) or
ContentType.IsDescendant(encoding('Invoice'))).

We use these statistics, since they subsume the set of rows specified by the query predicate, but we still have to apply the residual *ContentType.IsDescendant(encoding('Offer'))* to the statistics.

7.4 Refresh

Incremental maintenance of distribution statistics is prohibitively expensive. Therefore, the system tracks the number of changes (inserts, updates, deletes) to a column and uses heuristics to decide when statistics are considered stale and need to be refreshed. If the automatic statistics refresh is enabled on the database, the system will recalculate the statistics during query optimization whenever stale statistics are detected. This automatic update of statistics is typically sample based. The system uses a set of heuristics to determine the sample size. In case of filtered statistics the same logic applies. However, when determining the sample size, we take the filter selectivity into account. To guarantee a statistically meaningful sample size we "boost" the sample size by the filter selectivity.

Highly selective filter predicates could pose a problem, since we often would end up doing a full scan of the base table to achieve a statistically useful sample. In most cases we expect a filtered index associated with respective filtered statistics. In this case calculation of full scan statistics can be done very efficiently by scanning the filtered index instead of the unfiltered base table.

7.5 Beyond Flexible Schemas

Filtered statistics have applications beyond flexible schema implementations. One very common problem with distribution statistics is the assumption of statistical independence. As alluded to in the example above, estimating the selectivity of a more complex predicate is done by combining the estimates of individual basic predicates. All state of the art query optimizers assume statistical independence of the individual selectivities to calculate the combined selectivity. This assumption is often violated (sparse columns which contain values for only a small set of values in the discriminating column are a good example) and, in consequence, cardinality estimates can be arbitrarily bad. This problem can be alleviated with filtered statistics. In particular in cases where a column is discriminating and has only a small set of possible values, it is possible to create filtered statistics for some or all values (in the example above we would create filtered statistics for each possible *ContentType* value).

8. LARGE SCALE

In a normalized model for a heterogeneous collection metadata are partitioned between separate narrow tables. However, if one chooses a flexibility of wide tables, equivalent metadata of roughly the same size are tied to a single table. This triggers significant metadata scaling issues that have not been addressed by state of the art RDBMSs. A potentially large number of columns and dependant structures requires a cautious implementation of DDL routines and economical metadata loading in plan generation.

Note that, in order to execute a statement, one requires only a subset of columns from each considered table, i.e. the columns needed for plan generation. More precisely, this set contains all the columns referenced in the query, identified during algebrization, and these referenced in the structures used in the whole process. The key is to predict, which objects can be potentially useful and avoid loading unnecessary information or operating on large column collections in optimization. In particular, for SELECT queries, one loads columns from a query, interesting statistics, matchable and useful indices and materialized views. For DML, additional columns in affected dependant structures are also necessary. In case of wide tables, such selective approach usually decreases the number of columns visible in compilation by orders of magnitude.

Numerous statistics can be another source of deficiency in querying wide tables. Their number grows quickly, even if we assume only few for each logical subtable, and makes loading time a substantial factor. Moreover, although unfiltered statistics are loaded once, filtered ones may be scanned many times for various predicates considered in the optimization process. At the same time, for a particular query, one is interested in data distribution for a small number of interesting columns. Therefore, it is much more efficient to take an appropriate indexing strategy and seek directly to relevant statistics entries.

SQL Server ensures efficient handling of OLTP workloads for up to 300k columns, where at most 1k is non-sparse, 1k indices and 300k statistics. However, our experiments show that, in many cases, it scales well even beyond these limits.

9. PREVIOUS WORK

Flexible schemas deal with a problem of representing a heterogeneous and extensible collection of entities. This problem can be addressed with a classical vertical property bag, a solution proposed by Agrawal in [2]. Propositions for tuple reconstruction in this approach were covered in [2, 7].

Wide tables as a solution for heterogeneous collections were broadly discussed in [6] which contains a proposition in terms of storage, indexation and searching. Also, the idea of discovering a hidden schema by means of clustering was put forward. An interesting application of the design to store data extracted from semi-structured web pages was covered in [8]. The flexible schema scenario as a generalization of flat collections to hierarchical entity sets was introduced in [17]. Also, the importance of an easily extensible data model, in which instances are not strictly limited by class definitions was strongly emphasized in [14].

Interpreted storage for horizontal property bags was proposed in [3]. The concept was supported with an experimental comparison against positional and bitmap-only approaches implemented in PostgreSQL. In contrast to our method, non-null values are stored in a list, which leads to reduced performance. Column store is an alternative way of dealing with sparse tables ([1]).

Partial indexation was introduced by Stonebraker in [18, 19]. In general, the solution has been considered to index interesting values understood as non-default, exceptional, frequently queried ([16],[18]) or non-uniformly distributed ([15]). In result, further research ideas for costing and advising useful indices were described ([18],[15],[16]). For wide tables, a specific case of sparse indices, that target only non-null values, was recommended in [6]. Finally, a detailed presentation of filtered indices in SQL Server and their application to the flexible scenario can be found in [17].

The labeling scheme OrdPath was introduced in [13] as a compressed representation of node location in an XML document. An interesting application for efficient implementation of multilingual semantic matching was proposed in [12]. By analogy, we employed this convenient idea to class hierarchies in flexible schemas. Alternative labelings of XML data were discussed in [20, 11].

To the best of our knowledge there is no previous work on partial statistics per se. A problem can be handled by the concept of creating statistics on views ([9]) or an alternative approach discussed in [4] and [5]. However, due to a general character, they may suffer from an unnecessary overhead. We believe that our simpler solution suites better the considered use cases. Also, to some extent, the problem can be addressed by any model that takes into account correlation between columns, e.g. multi-dimensional histograms. Nonetheless, such approaches cannot benefit from the presence of a hidden schema and may behave less economical.

10. EXPERIMENTAL RESULTS

The goal of experiments is to verify performance of our package for a hierarchical flexible schema. A real world scenario is modeled by means of the prototypical content and collaboration application described in Sect. 2. Two classes of experiments have been performed. First, we measure the performance of each of our extensions in isolation towards standard relational solutions existing in SQL Server and, second, evaluate the performance impact of the whole integrated package.

In both classes of experiments we used a data generator referred here as SharepointLite. This specialized tool is able to create and populate databases statistically equivalent to the ones in deployed Sharepoint installations. As a result, synthesized sparse data gives insights on the behavior of real system without copying terabytes of database files or storing business sensitive data in a test environment. For the purpose of these experiments we used statistical data reflecting our largest internal Sharepoint database. A detailed description of data generation is a complex process and is beyond the scope of this paper.

In order to measure the performance characteristics of the proposed extensions, we used dynamic management views, a mechanism of the server, that provides concise statistics on query processing. We chose the execution time of a query as our main performance determinant for two reasons. First, it aggregates performance benefits from different sources, like query plan improvement, less number of disk operations etc. Second, execution time is the measure actually experienced by an end user. All experiments are run in single user mode, as we believe our extensions have insignificant impact on concurrent query execution and parallelism would only introduce additional test-to-test variation increasing measurement noise.

For a better picture of the systems behavior in the target environment, results are presented for both cold and warm data. The comparison shows how sensitive the solution is to existing caching mechanisms. For this reason, we use hardware with sufficient main memory to fit each of the used datasets in cache, namely, an 8-processor Intel Xeon MP 2.8 GHz with 4 GB of RAM.

10.1 Isolated Benchmarks

As a baseline experiment we implemented a traditional design that resulted in a database with multiple tables. There were four tables for each tested number of columns — two with regular and two with sparse columns. Of each two tables one was used to query and one to insert data. SQL Server supports only up to 1k regular columns, thus, we included tables with a large number of sparse columns without corresponding tables with the same number of regular columns.

SharepointLite was used to populate each of the query tables with the same set of 50k rows containing only 10 meaningful columns (of which around 80% values were non-null). In other words, each row represents an entry in a single logical subtable for a leaf content type, where 10 columns are meaningful. In a complete wide-table the remaining columns would be used in other rows to store data for other content types. Note that we decided to include some null values in meaningful columns as this reflects a real-world situation. Different column types were involved. The summary of these types together with a respective number of instances is listed in Tab. 4. The execution time was averaged over several runs for testing workloads.

Our first isolated test used a query selecting the first

Data type	Number of instances
DATETIME	1
FLOAT	3
NVARCHAR(255)	3
GUID	1
INT	2

Table 4: Column types used in tests

10 columns from a table. This is equivalent to retrieving all rows of a specified content type, when the meaningful columns for this list is known upfront. The results for cold data are shown in Fig. 3.

SELECT column1, . . . , column10 FROM T1

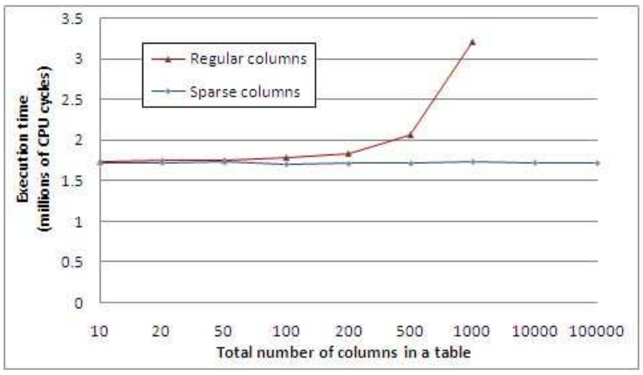


Figure 3: Querying individual columns (cold data)

The next test considered insertions of meaningful columns known upfront with data from a query table (Fig. 4). In order to limit potential noise from random I/O operations, we made sure that the data was loaded into memory before we accessed it.

INSERT INTO T2(column1, . . . , column10)
SELECT column1, . . . , column10 FROM T1

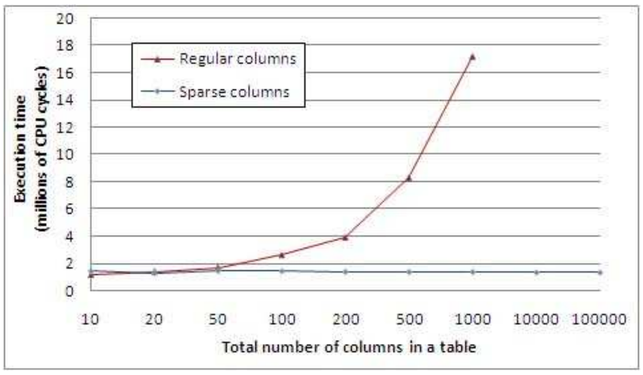


Figure 4: Inserting individual columns

Fig. 5 presents results for cold data, when a query selects all columns from each table. This reflects an operation in which the type of affected rows is not known, for example, administrative or generic operations, like backup, copy, move etc. In case of tables with sparse columns, the query resulted in one XML value of a sparse column set for each row, while in case of regular tables, a conventional matrix of all columns and rows was returned.

SELECT * FROM T1

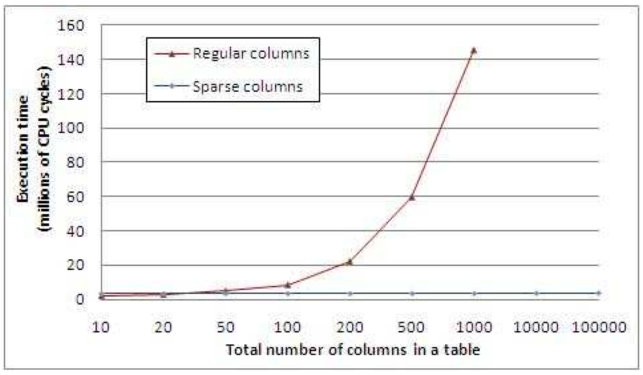


Figure 5: Querying all columns (cold data)

Accordingly, a complementary test looked at an insertion of data using all columns in a table (Fig. 6). For sparse columns it was performed by means of a sparse column set. Again, we made sure that the data was loaded in memory before we used it.

INSERT INTO T2
SELECT * FROM T1

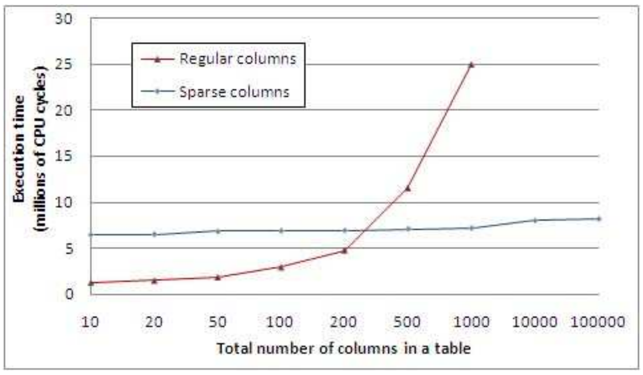


Figure 6: Inserting all columns

The last experiment for sparse columns repeated all the queries with warm data (Fig. 7). For a sparse column set, the results were similar to those obtained with cold data, but the query retrieving individual columns showed an interesting difference covered further in this section.

The results of the performed experiments are very appealing. Performance of practically all operations on sparse tables scale well, showing flat lines up to the tested maximum of 100k number of columns. Query and insert via individual columns have exactly the same execution time for all sizes of

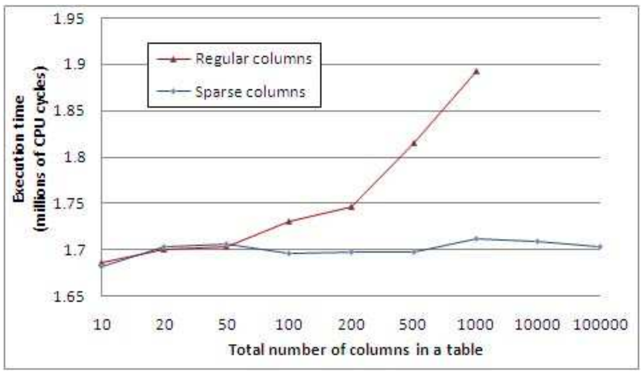


Figure 7: Querying individual columns (warm data)

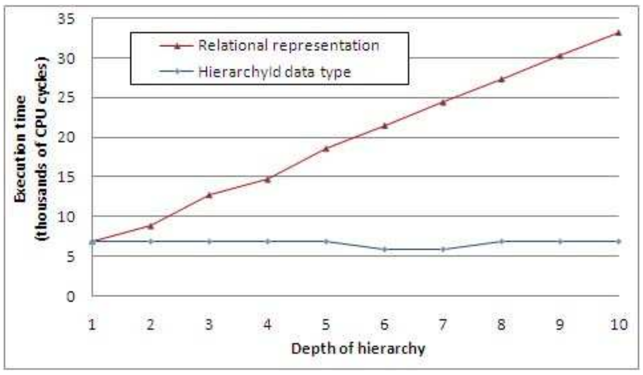


Figure 8: Querying hierarchy using relational representation and HierarchyId data type

tables, while column set operations present a slight dependency on it. We believe this is because translation of XML to/from sparse vector needs to access columns' metadata, which is stored in a B^+-Tree; despite the fact that the number of metadata accesses depends on the number of used columns only, the expected time for a single access depends logarithmically on the size of the tree. However, its impact on overall performance is marginal. In fact, retrieval of all columns from a 100k column table is only 9.5% slower than from a 10 column table.

Another interesting observation is that the storage benefits from the use of a sparse vector predominated the additional time spent on parsing when the percentage of null values in a row reached 70% or more. The only exception was the insert operation via column set, which has the tipping point placed at 97%. This could be caused by inefficiencies in the XML parser used to translate a sparse column set into a sparse vector. It is also worth noting that querying warm data via individual columns doesn't show any serious disadvantage from using sparse columns for any non-null to null ratio. Performance for sparse and regular columns stays within 1% from each other for data with less than 80% null values. Beyond this threshold sparse storage querying becomes much more efficient.

Benchmarks for filtered indices are not included in this paper, since the results presented in [17] refer to the same usage patterns and were performed for a similar synthetic data set. They prove the validity of filtered indexes to support fast data retrieval in a flexible schema environment. Nevertheless, the impact of filtered indexes on query performance is still evaluated for the integrated scenario.

The introduction of the HierarchyId data type plays a crucial role for data processing in flexible schemas. We believe that the most important property of this data type is its theoretical lack of dependency on the depth of the hierarchy. To measure this, we created a simple hierarchy with 10 levels and 100 elements at each level. Then, we implemented this hierarchy using a traditional relational representation of parent-child dependency and a recursive function testing if an element is an ancestor of another element. On the other hand, the same hierarchy was implemented by means of the HierarchyId data type. For cold data, we run a simple query counting all elements from a given level that are descendants of a certain top-level element (Fig. 8).

The results are very clear — not only for all queries operating on children (below level 1) the performance of the implementation with HierarchyId was definitely dominating over the traditional one and it was never slower. The results indicate that switching from a traditional model to the HierarchyId-based model does not introduce any performance risk.

10.2 Integrated Scenario

The second class of experiments refer to the integrated scenario that measures the aggregated impact of all implemented extensions on the performance of our prototypical schema. The data model follows our main example and is evaluated with queries that involve particular features.

SharepointLite was employed to synthesize a flexible schema with our prototypical type hierarchy (Fig. 1) containing 1k random user lists with exactly 1k different properties. Each type was associated with a respective HierarchyId value. We also created and populated a table to store a content type hierarchy in a parent-child relational format. Determining whether a given content type is a child of another given one was performed by a widely-known recursive T-SQL function IsAncestor(). Note that, in the traditional implementation, the same content type identifiers generated by HierarchyId were used. Then, two wide tables to store heterogeneous items were defined, one with regular and one with sparse columns. In the latter case, only 1,000 columns referring to class properties were sparse, whereas common columns remained marked as regular. Both tables were populated with the same 25k rows of random data.

As the last step of the test setup we created indices on both tables. We modeled a real world situation when some subset of the content types is queried very often and there's a need to index some of its properties. In our tests we decided to index the *CreateTime* property of all *Documents*. For the regular table approach, we created a regular index containing *CreateTime* and *ContentType* columns. For the sparse table scenario we created filtered indices on the same columns, but with a relevant predicate. We also created corresponding indices on the *ContentType* column itself, a regular one on the regular table and a filtered one on the sparse table.

We could see the first benefits from using the proposed extensions even before we ran any queries. The storage space savings were significant (Tab. 5).

Table 5: Sizes (in bytes) of database objects in regular and extended schemas

Object	Regular	Extended
Main table	88, 383, 488	15, 826, 944
Index on ContentType	1, 056, 768	106, 496
Index on ContentType and CreateTime	1, 261, 568	155, 648

The first query selects all properties stored in a given list from the type *Contacts* without considering subtypes and cannot use any index (Fig. 9):

```
SELECT
  FirstName, LastName, Phone1, ...
FROM T
WHERE
  ListId = LISTID AND
  ContentType = encoding('Contact')
```

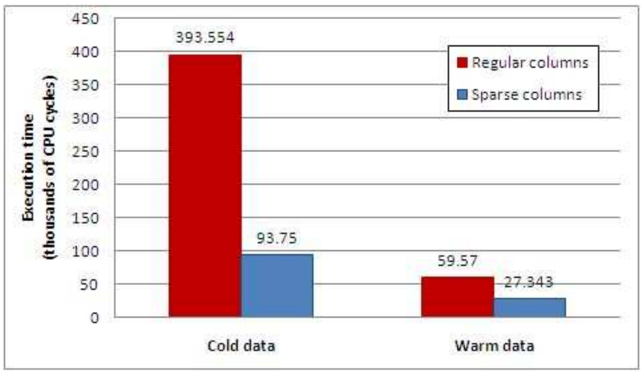


Figure 9: Query selecting all properties from all Contacts in a single list

The second query is similar, but the type *Invoice* is considered and indices are employed, regular indices for the regular table and filtered indices for the sparse one (Fig. 10).

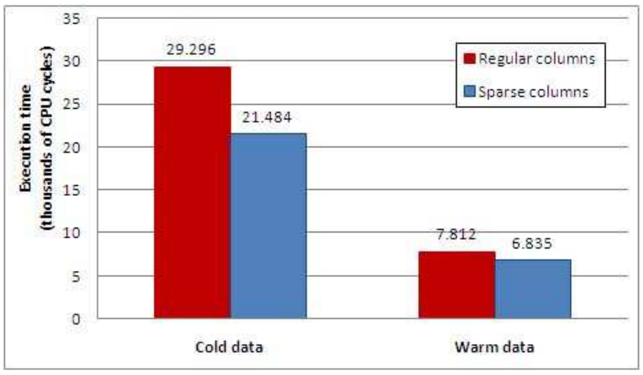


Figure 10: Query selecting all properties from all Invoices in a single list

The last query in our integrated tests requests all properties from the *Invoice* content type from the ten newest list items of this content type (including subtypes) in the table. For the table with our extensions we replaced the hierarchy function with a simple predicate, enabled by using HierarchyId. Moreover, we did not want the query to operate on indices only, thus, we included a predicate on the *FileSize* column to force the query plan to join and filter out some table rows. We chose this column, as it was not included in any index. The results can be found in Fig. 11.

```
SELECT TOP 10
  Author, Reviewer, FileSize, CreateTime, ...
FROM T
WHERE
  ContentType.IsDescendant(encoding('Invoice')) AND
  FileSize < 0
ORDER BY CreateTime
```

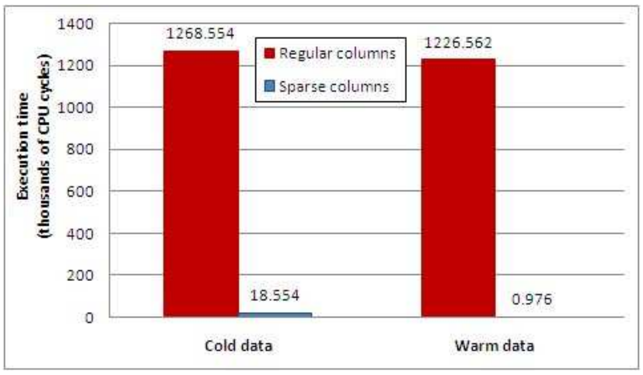


Figure 11: Query selecting ten newest Invoices (including subtypes) from all lists in the wide table

In the case of isolated benchmarks, we find the results of the integrated tests very encouraging. All queries we ran using our extensions show a significant improvement over the regular ones. This was especially evident for cold data, where sparse columns require much less disk reads. But even for warm data the table scan query was twice as fast.

The third query which incorporated all the described extensions proves that the framework is well-optimized for the considered scenario. In fact, we observe two orders of magnitude speed-up for cold data and three orders of magnitude one for warm data. Almost all the improvement is due to the combination of the HierarchyId data type and filtered indices. While HierarchyId allowed us to replace an expensive recursive function with a simple range predicate, the use of filtered indices eliminated the majority of data rows in a table from even being considered in the result. The combination of those two factors has brought a new quality to the performance of querying flexible schema data in the relational model.

11. CONCLUSIONS

In this paper we have presented a package of relational features, implemented in Microsoft SQL Server 2008, that support applications with flexible schema, and demonstrated their efficiency through a number of experiments.

In the relational approach, we view flexible schema by means of a potentially very wide table to store dynamic, hierarchical collections of entities. The shape of a hierarchy and granularity of collections defines a spectrum of data heterogeneity that goes from a few independent subtables embedded in the wide table, each with a strong schema and data homogeneity, all the way to individual rows that are unrelated to others in terms of structure or usage by the application. This utilization pattern emerges naturally in a number of applications, including content and collaboration.

The conceptual relational model can certainly cover the definition and manipulation of such a table, in terms of expressivity. However, typical RDBMS implementations assume both a limited number of columns and certain homogeneity in the data contents across a table, which align very well with traditional applications, but fail to efficiently support this new utilization pattern. Our focus in this paper has been on presenting a complete set of primitives that enable efficient and convenient manipulation of data on flexible schema tables. The package is made up of four related groups of features: (1) support for a large number of columns through sparse storage and column sets; (2) support for hierarchical relationships within a table; (3) support for efficient query processing for subtables through filtered indices and statistics; (4) economical loading of metadata to ensure independence of plan generation and execution from the total numbers of columns and structures defined in a table. Our experiments demonstrate the efficiency of these techniques for the target usage. The current server's limits are 300k sparse columns, 1k indices and 300k statistics in a table.

The proposed features extend the relational platform to efficiently cover a new, broad spectrum of applications while preserving traditional data management services.

12. ACKNOWLEDGEMENTS

We wanted to thank Jose Blakeley, Amit Shukla and Jennifer Beckmann for their valuable comments on our work.

13. REFERENCES

[1] D. J. Abadi. Column stores for wide and sparse data. In *CIDR*, Asilomar, CA, USA, 2007.

[2] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, page 58, Washington, DC, USA, 2006. IEEE Computer Society.

[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 263–274, New York, NY, USA, 2002. ACM.

[5] N. Bruno and S. Chaudhuri. Conditional selectivity for statistics on query expressions. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 311–322, New York, NY, USA, 2004. ACM.

[6] E. Chu, J. Beckmann, and J. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *SIGMOD*, pages 821–832, New York, NY, USA, 2007. ACM Press.

[7] G. G. Conor Cunningham, Cesar A. Galindo-Legaria. Pivot and unpivot: Optimization and execution strategies in an RDBMS. In *VLDB*, Toronto, Canada, 2004.

[8] T. C. A. D. J. F. N. Eric Chu, Akanksha Baid. A relational approach to incrementally extracting and querying structure in unstructured data. In *VLDB*, pages 1045–1056, 2007.

[9] C. A. Galindo-legaria, M. M. Joshi, F. Waas, and M.-c. Wu. Statistics on views. In *VLDB*, Berlin, Germany, 2003.

[10] C. A. Galindo-Legaria, S. Stefani, and F. Waas. Query processing for sql updates. In *SIGMOD*, pages 844–849, New York, NY, USA, 2004. ACM Press.

[11] T. Härder, M. Haustein, C. Mathis, and M. Wagner. Node labeling schemes for dynamic xml documents reconsidered. *Data Knowl. Eng.*, 60(1):126–149, 2007.

[12] A. Kumaran and P. Carlin. Multi-lingual semantic matching with ordpath in relational systems. *IEEE Data Eng. Bull.*, 30(1):44–56, 2007.

[13] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: insert-friendly xml node labels. In *SIGMOD*, pages 903–908, New York, NY, USA, 2004. ACM Press.

[14] J. Parsons and Y. Wand. Emancipating instances from the tyranny of classes in information modeling. *ACM Trans. Database Syst.*, 25(2):228–268, 2000.

[15] C. Sartori and M. R. Scalas. Partial indexing for nonuniform data distributions in relational dbms's. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):420–429, 1994.

[16] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 420–427, Washington, DC, USA, 1995. IEEE Computer Society.

[17] M. J. B. K. S. S. P. T. Srini Acharya, Cesar Galindo-Legaria. Filtered indices and their use in flexible schema scenarios. In *ICDE*, pages 903–908, Cancun, Mexico, 2008.

[18] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.

[19] M. Stonebraker, R. H. Katz, D. A. Patterson, and J. K. Ousterhout. The design of xprs. In *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases*, pages 318–330, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[20] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, New York, NY, USA, 2002. ACM Press.