Chapter 3

The Page Cube Model

In this chapter, we discuss a model for storage and retrieval of documents in an e-office during document production workflow with the context as the main binding element. This is a conceptual multi-dimensional model. The notion of a dimension provides a lot of semantic information, especially about the hierarchical relationship among its elements. The office documents are considered here as pages. We term the model as Page Cube (PC). A PC is a collection of registered pages of an office. Here pages are the main entities. A page has a profile, which describes the page and is defined by a set of attribute-value pairs. Registration of a page means adding and recording a new page to the page cube and assigning a unique page identifier, pid, to the new page. PC has two components: page space and page graphs. A preliminary version of the model appeared in [38].

3.1 The Page Space

The page space is an n-dimensional space defined by n orthogonal dimensions. Each dimension represents a theme and is defined by an attribute. An
attribute may have attributes and these in turn may have further attributes. Thus, the attributes of a dimension form a dimensional hierarchy. Therefore, we can say that the page space is defined by \( n \) orthogonal hierarchies. A page is represented in this space as a point, whose coordinate is an \( n \)-tuple. The main dimensions include the following but are not limited to:

- **Time**: It is a hierarchical dimension. The hierarchy is \( year.month.day.hour.min.sec \). The time dimension provides the time of creation of a page. It is a hierarchy with fixed depth.

- **Topic**: A page may be in one or more topics. A topic may have subtopics and a subtopic may be further classified. Thus it forms a topic hierarchy. Topic is a growing hierarchy.

- **Type**: A page may be of a type. A type may have subtypes and thus type forms a growing hierarchy. In time, a new type may be created under an existing leaf type. Generic to specific and to more specific types of pages in an office forms this hierarchy. For example, let the most general type of a page be *note*. A *note* may have subtypes *office order*, *report*, *comment* etc. Again, *office order* may have subtypes *appointment letter*, *termination letter*, *sanction order*, *circular*, *notice* etc. Similarly, *report* may have subtypes *inquiry*, *meeting minutes*, *field report* etc. and *comment* in turn may have subtypes *advice*, *remark*, *suggestion*, *ascent*, *descent* etc.

- **Category**: A page may belong to one of the three categories: *context*, *document* or *part*. The pages of category *part* are the elementary pages. A page of category *document* contains a set of links to the pages of category part. The links are ordered on the time of creation of the parts. The pages of category *documents* are the MPMSDs. A page of
category context contains a set of links to pages of category document. A DPW will have a context page associated with it.

- **Class**: Pages may be classified based on the themes of the content of the pages. The classes are rule, case, support, result-set and query. A page may contain rules on some topics, may be a part of a case of a workflow or may be a support page. A page may belong to more than one class. Content of a part of a case may be rules on some topics. For example, resolutions of the Board of Management (BoM) is the last part of a case of the BoM meeting workflow, while at the same time, it is a rule of concerned topics. The result of a query will also be a page containing links to the pages satisfying the query and such a page will be of class result-set. Finally, the queries are also stored in the page cube as pages, therefore query is another class.

- **User**: In DPW, user is an important dimension. A user belonging to an office may be the reviewer of some cases and therefore the author of some pages. For some cases, like the pages of category document or context, the author is the arbiter itself.

- **Domain**: The users of an office may belong to different domains of the office. Therefore, a page may be originated from a domain of an office. An office may have many domains and subdomains.

- **DPW**: In our model a page is produced in a DPW. This dimension gives the concerned workflow of a page.

- **State**: The pages in a DPW will be in one of the four states: active, reference, archived or burned.
This set of dimensions, common for all DPWs, is only a representative one. An office can identify more useful dimensions specific to the office concerned.

Figure 3.1: Page Cube

3.2 The Page Graphs

The pages of a PC are linked to a given page either implicitly or explicitly. Implicitly linked pages are those pages which satisfy a \textit{predicate} defined over the dimensional values of the pages. Explicitly linked pages are those pages which are linked by explicit hyperlinks. Thus, the pages, which are explicitly linked, form a directed graph, where the pages are the vertices and the hyperlinks are the edges. In addition to the implicit links provided by the
attributes of the dimensional hierarchy, the pages belonging to a dimension may be explicitly linked forming a dimensional graph (DG) of the concerned dimension. Each dimension will have one DG. Thus the page graph component of PC is a set of DGs.

Let $P$ be a PC, $G$ the set of all graphs in $P$, $V$ the set of all pages in $P$ and $E$ the set of all the directed edges of the graphs in $G$. In a dimension graph, pages are the vertices.

Let $G_d(V_d, E_d)$ be a digraph representing a dimension graph of dimension $d$, where $V_d \subseteq V$, $E \subseteq V$.

The vertices and the edges of $G_d(V_d, E_d)$ can be further classified into different kinds, based on the value of the dimension $d$.

Let $T_d^V$ and $T_d^E$ be the set of all different kinds of vertices and edges of $G_d(V_d, E_d)$. Then $V_d^\alpha$ is a set of vertices of dimension $d$ and of kind $\alpha$. Similarly $E_d^\beta$ is the set of edges of graph $G_d(V_d, E_d)$, of kind $\beta$. Again the edges may be classified based on the direction forward or reverse.

Accordingly, $E_d^\beta = E_d^{\beta+} \cup E_d^{\beta-}$, $E_d^{\beta+} \subseteq E_d^\beta$ is a set of forward edges and $E_d^{\beta-} \subseteq E_d^\beta$ is a set of reverse edges. Moreover, the edges in $E_d$ may be weighted, making $G_d(V_d, E_d)$ a weighted graph. The weight may be defined differently in different DGs. Linking of different kinds of pages by different kinds of edges in $G_d(V_d, E_d)$ is subject to the satisfaction of a set of constraints $C_d$. The act of linking two pages by creating a pair of conjugate edges is termed as plugging.

### 3.2.1 Pages

Pages are the main entities to deal with in a PC. Therefore, we should have a clear understanding of a page and its constituent elements in the context of the page space and page graph components of PC. To define a page, we
first define a few sets as follows:

Let $S$ be a countably infinite set of strings,

$A \subseteq S$ be a set of variables called attributes,

$C \subseteq S$ be a set of allowed values for the attributes in $A$.

From the preceding discussion, we define an edge as follows:

**Definition 3.1** An edge $e \in E$ is a $6-$tuple, $(source, target, graph, kind, direction, weight)$, where $source$ is the source page, $target$ is the target page, $graph \in G$ is the concerned dimensional graph, $kind$ is the kind of the edge, $direction$ is the the direction of the edge: either forward($+$) or backward($-$)} and weight is the weight of the edge.

With these definitions let us now define a page as follows:

**Definition 3.2** A page $p \in V$ is defined as a quadruple $p = (B, X, L_+, L_-)$, $B$ is the profile, $X \subseteq S$ is a set of strings defining the content of the page, $L_+ \subseteq E$ is the set of forward edges, $L_- \subseteq E$ is the set of reverse edges, where for every edge $e \in E$ $L_+ \cup L_-$, $e.source = p$.

$e.source = p$ in the definition 3.2 is a format of writing the value of the source attribute of tuple $e$ is $p$.

**Definition 3.3** A profile $B$ is defined as a $2$-tuple, an attribute and a list of values, $B = \{(a, \{v_1, v_2, \cdots, v_n\}) | a \in A, v_i \in C, i = 1, 2, 3, \cdots, n\}$

A page is a point in the multi-dimensional page space of a PC. Its coordinates are defined in the profile of the page. The profile of a page contains the values of the attributes of the page. The attributes of a page may be dimensional and non-dimensional attributes. A dimensional attribute contains the values of the page for a dimension of the PC. A non-dimensional attribute contains the values other than the dimensional values. For example, *type, topic, category* etc. are the dimensional, whereas
pageId, signature, size etc. are the non-dimensional attributes of a page. The kind of a vertex (which is a page) of a DG is stored in a page as a dimensional attribute. Similarly graph, kind and direction of an edge of a DG are stored as attributes of an edge.

A page of a particular type may be on more than one topic. In that case, a page may have multiple n-tuple coordinates. That is the reason, a distinct pageId is chosen as the logical address of a page. Otherwise, the n-tuple coordinate would have been an ideal logical address of a page. Without any loss of generality, we can use the time of creation of a page as the unique pageId, since in our scheme the timestamp will be given by a central arbiter [37].

3.2.2 The Category Graph

Among the different common dimensions discussed in section 3.1, the dimension category is a very important dimension. For a DPW, the construction of the dimensional graphs for category is mandatory. Therefore, we shall discuss the category graphs in detail in this section.

A page may belong to one of the three categories: part, document and context, as discussed earlier. A part may cite other parts during production of the part. Similarly a part may be cited in many parts. Therefore, pages of category part are linked by cite kind of edges. Moreover, a part itself may consist of multiple parts due to revision of the part from time to time, as discussed in detail in section 3.2.3. Therefore, a part may be linked by splitPart or revisePart kinds of edges. A document may plug many parts and a part may be plugged in many documents. Therefore, pages of category document and part are linked by docPart kind of edges. A context may plug many documents and a document may be plugged in many contexts.
Therefore, pages of category context and document are linked by conDoc kind of edges. It certain cases, a context may be plugged to another context by conCon kind of edges. For example, in addition to the documents specific to a case, the context of a case includes the context of the DPW. So, the context page of a case plugs the context page of the corresponding DPW.

Let \( G_{\text{category}}(V_{\text{category}}, E_{\text{category}}) \) be the dimensional graph for the dimension category. Let \( T^V_{\text{category}} = \{\text{part}, \text{document}, \text{context}\} \) be the set of kinds of vertices, \( T^E_{\text{category}} = \{\text{cite}, \text{splitPart}, \text{revisePart}, \text{docPart}, \text{conDoc}, \text{conCon}\} \) the set of kinds of edges, 
\[ V_{\text{category}} = \{V_{\text{part}}, V_{\text{document}}, V_{\text{context}}\} \]
where \( V_{\text{part}} \) is a set of vertices of kind part, \( V_{\text{document}} \) is a set of vertices of kind document and \( V_{\text{context}} \) is a set of vertices of kind context.

\[ E_{\text{category}} = \{E_{\text{cite}}, E_{\text{splitPart}}, E_{\text{revisePart}}, E_{\text{docPart}}, E_{\text{conDoc}}, E_{\text{conCon}}\} \]
where \( E_{\text{cite}} \) is a set of edges of kind cite, \( E_{\text{splitPart}} \) is a set of edges of kind splitPart, \( E_{\text{revisePart}} \) is a set of edges of kind revisePart, \( E_{\text{docPart}} \) is a set of edges of kind docPart, \( E_{\text{conDoc}} \) is a set of edges of kind conDoc and \( E_{\text{conCon}} \) is a set of edges of kind conCon.

A page may be plugged to another page at some time and may be unplugged at some other time. The weight of an edge is a 2-tuple \((t_+, t_-)\), where \( t_+ \) is the time of plugging and \( t_- \) is the time of unplugging. The significance of the temporal weight is that an edge remains active from the moment it is plugged till it is unplugged, that is, during the period defined by \( t_+ \) and \( t_- \). Therefore, edges in \( G_{\text{category}}(V, E) \) are persistent in nature. Unplugging does not delete the conjugate pair of edges, it only modifies their weights. The temporal weights of edges keep the history of the plugging of pages and are used in the retrieval of a stage of a page at a particular time. It will be discussed later. Since there is a possibility of plugging as well as unplug-
ging more than once between the same pair of vertices, parallel edges with
different weights may exist.

The graph $G_{category}(V_{category}, E_{category})$ is constructed by plugging the pages
subject to the satisfaction of the following constraints:

- **Citation Constraints**: A page $p_i$ cites another page $p_j$ iff the following
  conformability conditions on operands are satisfied:
  (i) $p_i \in V_{part}$ and $p_j \in V_{part} \cup V_{document}$
  (ii) $p_i, p_j \in P$.
  The significance of the first restrictions is that a page of category part
can cite another page of category either part or document. According
to the second restriction, actual plugging takes place only after regis-
tration of the new page in the page cube.

- **DocumentPart Constraints**: A page $p_j$ is plugged to page $p_i$ iff the
  following conformability conditions are satisfied:
  (i) $p_i \in V_{document}$ and $p_j \in V_{part}$
  (ii) $p_i$ is in active state.
  (iii) $p_i, p_j \in P$.
  A page of category part can be plugged to a page of category document
  in active state only. When a new page of category document is to be
  created, first a null document is created then other pages are plugged.

- **ContextDocument Constraints**: A page $p_j$ is plugged to page $p_i$ iff
  the following conformability conditions are satisfied:
  (i) $p_i \in V_{context}$ and $p_j \in V_{document}$
  (ii) $p_i$ is in active state
  (iii) $p_i, p_j \in P$
• **PartPart Constraints**: A page $p_j$ is plugged to page $p_i$ iff the following conformability conditions are satisfied:

(i) $p_i, p_j \in V_{part}$

(ii) $p_i$ is in active state

(iii) $p_i, p_j \in P$

• **ContextContext Constraints**: A page $p_j$ is plugged to page $p_i$ iff the following conformability conditions are satisfied:

(i) $p_i, p_j \in V_{context}$

(ii) $p_i$ is in active state

(iii) $p_i, p_j \in P$

### 3.2.3 Revision of a Page

Revision of a page is applicable only to the pages of category *part*. For certain classes of pages of category *part*, splitting of a part may be necessary. For example, let us consider a page of category *part* and class *rule*, containing rules on some topics. These rules may be revised from time to time. Due to revision only a certain portion of the content of the page may be changed in the revised version and the remaining portions of the content is intact. There are two ways to take care of such time varying parts. In the first approach, a new version of the part is created in a new page. This page is plugged to the document page and the page containing the old version is unplugged. The disadvantage of this approach is that for a minor revision in the content of the page, which is very common in an office, a major unrevised portion of the content is replicated in the new version of the page. In the second approach,
an authorized user selects a page to be revised and marks a portion to be replaced by a new portion. The page is split into a number of portions and placed in separate newly created pages. One of these pages contains the revised portion, while the other pages (there can be 0 to 2 more such pages depending on whether the revised portion covers the entire original page, is in the middle of the page, or is at one end of the page), contain unrevised portions. All these pages are plugged to the original page and become its children, so to speak. The original page now has no content in it. Instead, its content is to be recovered from the contents in its child pages based on the given time of recovery. A page of category part which has child pages may be considered to be a compound part. We shall refer to the parts in these child pages as portions to ease the discussion. The algorithm to revise a part is given below as algorithm RevisePart. One aspect that needs to be pointed out is that, as a result of this scheme, there is a need to distinguish between the time of creation of a page and the time of creation of the content of a page. The former is represented by the pageId, while the later is stored as the dimensional attribute time in the profile of a page. A newly created child page may have in it very old content.

**ALGORITHM RevisePart**($pid, begin, end, new\_portion$)

Algorithm to revise a page $p$ of category part

**INPUT:**

$pid$: pagId of $p$ to be revised

$begin$: beginning position of the portion in $p$

$end$: ending position of the portion in $p$

$new\_portion$: content of new revised portion

**OUTPUT:**

$status$: 1 if successfully revised, 0 otherwise
ASSUMPTIONS:
\(status = 0\) initially, when the algorithm is called

NOTATIONS:
\(t_p\): time of creation of the content of \(p\)
\(null\): a constant with value 0
\(now\) is also a timestamp, which represents the present time
\(timeStamp()\): a function which returns the present time
\(endOfPage\): end position of a page, taking beginning position as 0

BEGIN

Begin Case

\textbf{step 1: case 1: } begin = 0 and end = endOfPage

// full page is to be replaced

\textbf{step 1.2: } Register a new page with \(\text{pid}_{old}\) and category \(part\)

Copy the content of \(\text{pid}\) to \(\text{pid}_{old}\);

Set \(X\) of \(\text{pid}\) to \(null\) //content of \(\text{pid}\) is removed

Set the time of creation of content of \(\text{pid}_{old}\) to \(t_p\)

\textbf{step 1.3: } Register a new page of category \(part\) with \(\text{pid}_{new}\)

containing \(new\_portion\)

\textbf{step 1.4: } Set \(now = timeStamp()\);

Plug \(\text{pid}\) to \(\text{pid}_{old}\) with a pair of edges of kind \(revisePart\),

where \(plugtime = t_p\) and then unplug it with \(unplugtime = now\);

Plug \(\text{pid}\) to \(\text{pid}_{new}\) with a pair of edges of kind \(revisePart\),

where \(plugtime = now\) and \(unplugtime = null\);

Set the time of creation of content of \(\text{pid}_{new} = now\)

\textbf{step 1.5: } return \((status = 1)\)

\textbf{step 2: case 2: } begin = 0 and end < endOfPage

// top portion of the page is to be replaced
step 2.1: Split pid into two portions $p_1$ and $p_2$ at position $end$
Register $p_1$ and $p_2$, as new pages of category $part$
with pageIds $pid_1$ and $pid_2$ respectively

step 2.2 Copy the portion from $begin$ to $end$ to $pid_1$ and
from $end + 1$ to $endOfPage$ to $pid_2$
Set $X$ of $pid$ to null //content of $pid$ is removed
Set the time of creation of content of $pid_1$ and $pid_2$ to $t_p$

step 2.3: Plug $pid$ to $pid_1$ with a pair of edges of kind $splitPart$,
where $plugtime = t_p$ and $unplugtime = null$

step 2.4: Plug $pid$ to $pid_2$ with a pair of edges of kind $splitPart$,
where $plugtime = t_p$ and $unplugtime = null$

step 2.5: RevisePart($pid_1$, $begin = 0$, $end = endOfPage$, new portion)

step 3: case 3: $begin > 0$ and $end = endOfPage$
//bottom portion of the page is to be replaced

step 3.1: split $pid$ into two portions $p_1$ and $p_2$ at position $begin$
Register $p_1$ and $p_2$, as new pages of category $part$
with pageIds $pid_1$ and $pid_2$ respectively

step 3.2 Copy the portion from 0 to $begin - 1$ to $pid_1$ and
from $begin$ to $endOfPage$ to $pid_2$
Set $X$ of $pid$ to null //content of $pid$ is removed
Set the time of creation of content of $pid_1$ and $pid_2$ to $t_p$

step 3.3: Plug $pid$ to $pid_1$ with a pair of edges of kind $splitPart$,
where $plugtime = t_p$ and $unplugtime = null$

step 3.4: Plug $pid$ to $pid_2$ with a pair of edges of kind $splitPart$,
where $plugtime = t_p$ and $unplugtime = null$

step 3.5: RevisePart($pid_2$, $begin = 0$, $end = endOfPage$, new portion)

step 4: case 4: $begin > 0$ and $end < endOfPage
//middle portion of the page is to be replaced

**step 4.1** Split *pid* into three portions *p₁*, *p₂* and *p₃* at positions *begin* and *end*

Register *p₁*, *p₂* and *p₃*, as new pages of category *part* with pageIds *pid₁*, *pid₂* and *pid₃* respectively

**step 4.2** Copy the portion from 0 to *begin* - 1 to *pid₁*,

from *begin* to *end* to *pid₂* and from *end* + 1 to *endOfPage* to *pid₃*

Set *X* of *pid* to *null* //content of *pid* is removed

Set the time of creation of content of *pid₁*, *pid₂* and *pid₃* to *tₚ*

**step 4.3** Plug *pid* to *pid₁* with a pair of edges of kind *splitPart*,

where *plugtime* = *tₚ* and *unplugtime* = *null*

**step 4.4** Plug *pid* to *pid₂* with a pair of edges of kind *splitPart*,

where *plugtime* = *tₚ* and *unplugtime* = *null*

**step 4.5** Plug *pid* to *pid₃* with a pair of edges of kind *splitPart*,

where *plugtime* = *tₚ* and *unplugtime* = *null*

**step 4.6** RevisePart(*pid₂*, *begin* = 0, *end* = *endOfPage*, *newportion*)

End Case

END

All the four possible cases of revision of a part are taken care of in the algorithm RevisePart(). In case 1 (step 1), entire part is revised by a new part. In case 2 (step 2), case 3 (step 3) and case 4 (step 4) revision of top, bottom and middle portions of a part are taken care of. A page is created first, and then may be plugged to one or more pages. Therefore the time of creation of a page is always less than or equal to the time of plugging the page to other pages. When a page is split to multiple portions, the portions of a page collectively contain the content of the split page. Therefore, the time of creation of the contents of the portions are equal to the time of creation of the original part. When a state of the page at a particular time
is retrieved, the portions of the page not revised are to be included intact. The portions are to be retrieved with respect to the time of plugging and unplugging. Therefore, if the plug time of the portions, that are unrevised and the old versions of the revised portions, are set to the \( t_p \), the time of creation of the content of the part to be revised, then the retrieval will be correct. The retrieval of a state of a page at a particular time is discussed in the next section.

### 3.2.4 Retrieval of a Page

Let us first discuss how to retrieve a page of category *part*, which may have multiple portions, and some portions may be revised. We discuss below an algorithm \( \text{GetPart()} \) to retrieve the state of a page of category *part* at time \( t \). The algorithm returns a list of pageIds, and the concatenation of the contents of these pageIds gives the state of the part at time \( t \). Since a portion is registered as a page with a unique pageld and the portions are registered in order, therefore pagelds of children nodes of a parent node gives the order. A portion may have further portions as children. A part may have three types of edges: *cite*, *splitPart* and *revisePart*. A leaf node is identified as a node having no *splitpart* as well as *revisePart* kinds of edges. An internal node will have either *splitPart* or *revisepart* kind of edges. The leaf nodes which satisfy the temporal conditions on time \( t \), given in the algorithm \( \text{GetPart()} \), constitute the state of a part at time \( t \) from the tree. When \( t = \text{now} \), we get the current state of a part.

**ALGORITHM GetPart(pid, t)**

Algorithm to retrieve the state \( p' \) of a page \( p \) of category *part*, at time \( t \)

**INPUT :**

\( pid \) : pageld of part \( p \) to be retrieved
\( t \): time which defines the state of \( pid \)

**OUTPUT:**

\( pidList^t \): list of pageids, in order, which constitute the part \( pid \) at time \( t \)

**ASSUMPTIONS:**

\( pidList^t \) is initially \( null \)

**NOTATIONS:**

\( pid_1 \| pid_2 \): concatenation of \( pid_1 \) and \( pid_2 \)

\( null \): a constant with value 0

\( now \) is a timestamp, which signifies the present time

\( element.attribute = value \): the structure is a short form of writing the value of the attribute of an element

**BEGIN**

**step 1:** Retrieve page \( p \) with \( pid = pid \)

**step 2:** From \( p \), form a set of forward edges of kind \( splitPart \)

\[
L^s_+ = \{ e^s \mid e^s.source = pid \text{ and } e^s.kind = splitPart \\
\text{ and } e^s.dir = + \text{ and } e^s.t_+ \leq t \text{ and } (e^s.t_- > t \text{ or } e^s.t_- = null) \}
\]

**step 2.1:** Form \( pidList^s \) of pageids of pages of category \( part \) from \( L^s_+ \)

\[
pidList^s = \{ p_s \mid e^s.target = p_s, \text{ where } e^s \in L^s_+ \}
\]

**step 2.2:** if \( pidList^s \) is not \( null \) then sort pageids in \( pidList^s \)

in ascending order of pageids

**step 3:** From \( p \), form a set of forward edges of kind \( revisePart \)

\[
L^r_+ = \{ e^r \mid e^r.source = pid \text{ and } e^r.kind = revisePart \\
\text{ and } e^r.dir = + \text{ and } e^r.t_+ \leq t \text{ and } (e^r.t_- > t \text{ or } e^r.t_- = null) \}
\]

**step 3.1:** Form \( pidList^r \) of pageids pages of category \( part \) from \( L^r_+ \)

\[
pidList^r = \{ p_r \mid e^r.target = p_r, \text{ and } e^r \in L^r_+ \}
\]

**step 3.2:** if \( pidList^r \) is not \( null \) then sort pageids in \( pidList^r \)

in ascending order of pageids

68
step 4: if \( pidList^S = null \) and \( pidList^R = null \) then
\[ pidList^t = pidList^t || pid \]

step 4.1: else if \( pidList^S \neq null \) then
for each \( p_s \in pidList^S \)
\[ pidList^t = pidList^t || GetPart(p_s, t) \]

step 4.2: else if \( pidList^R \neq null \) then
  for each \( p_r \in pidList^R \)
  \[ pidList^t = pidList^t || GetPart(p_r, t) \]
endif

step 5: return(\( pidList^t \))

END

The algorithm \( GetPart() \) gives the state of a part \( pid \) at time \( t \). Once we take care of pages of category part retrieval of a page of category document is a simple one, similar to \( GetPart() \). The algorithm \( GetDocument() \) is given below:

**ALGORITHM GetDocument(\( pid, t \))**

Algorithm to retrieve the state \( p^t \) of a page \( p \) of category document at time \( t \)

**INPUT:**

\( pid \) : pagId of document \( p \) to be retrieved
\( t \) : time which defines the state of \( pid \)

**OUTPUT:**

\( pidList^t \) : list of pageids, in order, which constitute the document \( pid \) at time \( t \)

**ASSUMPTIONS:**

\( pidList^t \) is initially null
NOTATIONS:

$\text{pid}_1||\text{pid}_2$ : concatenation of $\text{pid}_1$ and $\text{pid}_2$

$null$ : a constant with value 0

$now$ is also a timestamp, which signifies the present time

$element.attribute = value$ : the structure is a short form of writing the value of the attribute of an element

BEGIN

step 1: Retrieve page $p$ with pageId = $\text{pid}$

step 2: From $p$, form a set of forward edges of kind $\text{docPart}$

$\begin{align*}
L_+^s &= \{ e^s | e^s.source = \text{pid} \text{ and } e^s.kind = \text{docPart} \\
&\text{ and } e^s.dir = + \text{ and } e^s.t_+ \leq t \text{ and } (e^s.t_- > t \text{ or } e^s.t_- = \text{null}) \} 
\end{align*}$

step 2.1: Form $\text{pidList}^s$ of pageIds of pages of category $\text{part}$ from $L_+^s$

$\text{pidList}^s = \{ e^s.target = p_s, \text{ where } e^s \in L_+^s \}$

step 2.2: if $\text{pidList}^s \neq \text{null}$ then

sort pageIds in $\text{pidList}^s$ in ascending order of pageIds

for each $p_s \in \text{pidList}^s$

$\text{pidList}^t = \text{pidList}^t|| \text{GetPart}(p_s, t)$

endif

step 3 : return($\text{pidList}^t$)

END

Retrieval of a page of category $context$ is similar to $\text{GetDocument()}$. Since, production of context is the subject matter of Chapter 4, it is discussed in detail there.
3.2.5 An Example

Let us consider again the travel plan workflow discussed in Chapter 1 and 2. Portion of a category graph for such a workflow is shown in figure 3.2. The node $e$ is the document that is under examination. This document has four parts, $i$, $j$, $k$ and $l$ corresponding to employee $A$'s application, reviewers $B$, $C$ and $D$'s comments respectively. The document $e$ has a case context and this is node $a$. Node $a$ points to document $d$, which is being used by $A$ as it is her leave account. The context $a$ also points to the DPW context $b$. The context $b$ includes document $f$, which is an earlier case acting as a precedent for this DPW and it has a case context $c$. The context $b$ also includes document $g$, which is the set of leave rules. This rules were originally in node $q$, but a portion got revised. As a result, nodes $q_1$, $q_2$ and $q_3$ were created. $q_1$ and $q_2$ contains the first and the last part of the rules while $q_2$ points to $q_21$ and
q22. q21 contains the pre-revised portion and q22 the revised portion of the rules. q21 was unplugged when q22 was plugged. The rules currently are therefore contained in q1, q22 and q3. A link emanating from node r to q22 is a citation of the rules in q22 by applicant A. There is another citation link from node n to q shown. Here the entire set of rules are being cited.

3.3 Query Languages

A query is an expression denoting a set of pages described by a formula \( \phi \) of the form \( \{p|\phi(p)\} \). A query language provides a user with a means of expressing questions in the form that can be handled by the model enabling the model to answer the questions asked in a reasonable time. In this section we describe two equivalent query languages: The first language is Page Algebra (PA), a procedural language which uses specialized operators on the sets of pages to specify queries and a Page Structured Query Language (PSQL), a user-friendly pseudo-natural language with a simple means for expressing queries using a natural language form. The languages are similar to Relational Algebra and SQL respectively.

3.3.1 Path Expression

A Path Expression (PE) defines a path from one node in the graph to another in terms of intermediate node and edge labels. PEs in graphs are used in navigation oriented queries. In our model, navigation in the dimensional graphs is a common feature for the queries. Moreover, the dimensions of the PC are also hierarchical. Therefore, values of the attributes can be expressed as PEs. Details of use of path expressions in document databases is given in [14]. We follow the simplified PE discussed in [35]. The standard wildcat
character "*" is used in a path expression to signify all successors of a node in the dimension hierarchy as well as in the dimensional graph. The standard "." operator, commonly used to denote attributes of a relation in the relational model can now be cascaded to express a listed path. In addition a ".." operator is introduced, which is used to construct an abbreviated path from a listed path. For example, a fully listed path expression is $p_1.p_2.p_3.p_4.p_6.p_7$, where $p_i, i = 1, 7$ is a pageId. An abbreviated path $p_1..p_7$ means that there is a path between $p_1$ and $p_7$, but the actual path itself is not of significance. Formally, the above expression evaluates to:

$$\exists x PATH(x) \cap p_1.x.p_7$$

".." and "*" takes care of the don't care conditions in a PE. Details of PEs is available in[35].

### 3.3.2 Page Algebra

Page Algebra(PA) is an operator based query language for querying pages from a PC. It is an extension of relational algebra. PA is defined in terms of a special set operators that map one or more sets of pages to a new set of pages. Every PA expression $E$ represents a set of pages. The main operators are as follows:

- **Selection ($\sigma$):** The selection operation $\sigma_\gamma E$ extracts a subset of pages from an input set $E$ that satisfies the selection condition $\gamma$

  $$\sigma_\gamma E = \{p | p \in E \land \gamma\}$$

- **Plug ($\times$):** Given two PA expressions $E_1$ and $E_2$, the expression $E_1 \times E_2$ reproduces the pages belonging to $E_1$ with forward edges to the pages belonging to $E_2$ and the pages belonging to $E_2$ with backward edges to pages belonging to $E_1$. 

  73
Let $E_1 = \{p_1, p_2\}$ and $E_2 = \{p_3, p_4\}$. $E_1 \times E_2$ produces the pages with forward (+) and backward (-) edges. For example, for the category graph, with graph id $g$ has context to document(cd), document to part(dp), part to part(pp) kinds of edges. Accordingly, the forward and backward edges may be qualified. If $E_1$ contains pages of category document and $E_2$ contains parts, then the qualified edges due to $E_1 \times E_2$ will be

$\begin{align*}
P_1 &= (p_1, \{(p_3, g, (t_+,), dp, +), (p_4, g, (t_+,), dp, +)\}) \\
P_2 &= (p_2, \{(p_3, g, (t_+,), dp, +), (p_4, g, (t_+,), dp, +)\}) \\
P_3 &= (p_3, \{(p_1, g, (t_+,), dp, -), (p_2, g, (t_+,), dp, -)\}) \\
P_4 &= (p_4, \{(p_1, g, (t_+,), dp, -), (p_2, g, (t_+,), dp, -)\})
\end{align*}$

- **Unplug ($\div$)**: This is the reverse operation of plug. Given two PA expressions $E_1$ and $E_2$, the expression $E_1 \div E_2$ reproduces the pages belonging to $E_1$ deactivating forward edges to the pages belonging to $E_2$ and also reproduces the pages belonging to $E_2$ deactivating the backward edges to pages belonging to $E_1$. Unplug does not necessarily remove the edges physically. It simply modifies the weights of the edges by incorporating the time of unplugging ($t_-$). After unplugging the pages of $E_1$ and $E_2$ change to the following:

$\begin{align*}
P_1 &= (p_1, \{(p_3, g, (t_+, t_-), dp, +), (p_4, g, (t_+, t_-), dp, +)\}) \\
P_2 &= (p_2, \{(p_3, g, (t_+, t_-), dp, +), (p_4, g, (t_+, t_-), dp, +)\}) \\
P_3 &= (p_3, \{(p_1, g, (t_+, t_-), dp, -), (p_2, g, (t_+, t_-), dp, -)\}) \\
P_4 &= (p_4, \{(p_1, g, (t_+, t_-), dp, -), (p_2, g, (t_+, t_-), dp, -)\})
\end{align*}$
• **Path Selection** ($\circ$): This is basically a navigational operation, where $\circ \in \{.,\ldots\}$. Given a PA expression $E$ and a PE $P$, $E \circ P$ returns the set of pages obtained after traversing the path $P$ from each of the pages in $E$ [35].

• **Union** ($\cup$): Union is the normal set union operation. Given two PA expression $E_1$ and $E_2$. The result of $E_1 \cup E_2$ is a set of pages defined as

$$E_1 \cup E_2 = \{p| p \in E_1 \lor p \in E_2\}$$

• **Intersection** ($\cap$): Intersection is the normal set intersection operation. Given two PA expression $E_1$ and $E_2$. The result of $E_1 \cap E_2$ is a set of pages defined as

$$E_1 \cap E_2 = \{p| p \in E_1 \land p \in E_2\}$$

• **Difference** ($\setminus$): is the normal set difference operation. Given two PA expression $E_1$ and $E_2$. The result of $E_1 \setminus E_2$ is a set of pages defined as

$$E_1 \setminus E_2 = \{p| p \in E_1 \land p \notin E_2\}$$

**Examples**

Let us look at some examples to illustrate the different operators of PA.

1. Find from the page cube $P$, all the documents containing rules on special casual leave and include them in the context of the dpw $w$. In PA it can be expressed as

$$\text{category} = \text{"context" \& dpw = "w"}(P)$$
2. Find from the page cube $P$, all the documents containing rules on special casual leave and exclude them from the context of the dpw $w$.

\[ \sigma_{\text{category} = \text{context} \land \text{dpw} = w}(P) \]  
\[ \ominus \]  
\[ \sigma_{\text{category} = \text{document} \land \text{class} = \text{rules} \land \text{topic} = \text{leave.casual.special}}(P) \]

3. Find from page cube $P$, all the parts included in the context of the dpw $w$ and signed by user $u$.

\[ (\sigma_{\text{category} = \text{context} \land \text{dpw} = w}(P)) \ominus (\sigma_{\text{category} = \text{part} \land \text{user} = u}(P)) \]

This expression returns all parts signed by $u$ if there exists a path from the context of $w$ to the part.

4. Find from the page cube $P$ all office orders on leave which are neither sanction orders nor the circulars.

\[ (\sigma_{\text{type} = \text{..office.order..order.*} \land \text{topic} = \text{..leave.**}}(P)) \]

\[ \ominus \]  
\[ (\sigma_{\text{type} = \text{..office.order.sanction..order.sanction.*} \land \text{topic} = \text{..leave.**}}(P)) \]

\[ \cup \]  
\[ \sigma_{\text{type} = \text{..office.order.notice..order.notice.*} \land \text{topic} = \text{..leave.**}}(P)) \]

### 3.3.3 Page Structured Query Language

Here we present a language, called PSQL, for interactively processing queries on pages. PSQL is an extended version of SQL. The primary motivation behind such a language is to provide users of database systems with a simple
means for expressing queries using a natural language form. Standard SQL
deals with flat tables and the result set of an SQL query is also a table. In SQL
the main retrieval operation is the SELECT operation. To accommodate this
feature in PSQL the SELECT clause will have the mechanism to allow the
creation of composite page from the constituent pages. The resultant page
of a query is basically a multi-part page which contains hyperlinks(edges) to
the constituent pages.

Examples

Let us consider some queries expressed in PSQL.

1. Find all the precedents of the workflow \( w \) from the page cube \( P \).

```sql
SELECT page q
FROM   cube P
WHERE  q.profile.dimension.category = document
        and q.profile.dimension.class = case
        and q.profile.dimension.dpw = w
        and q.profile.dimension.state = reference
```

2. Find all the parts of the precedents of the workflow \( w \) from the page
cube \( P \).

```sql
SELECT page q
FROM   cube P, P.graph.category g
WHERE  q.profile.category = part
```
and g.dpw = w
and g.edge.target = q
and g.edge.kind = docpart
and g.edge.dir = +
and g.edge.source IN
(SELECT page p
FROM cube P
WHERE p.profile.dimension.category=document
and p.profile.dimension.class=case
and p.profile.dimension.dpw=w
and p.profile.dimension.state=reference)

3. Find all documents from a page cube $P$ containing rules on casual leave or on travel abroad produced after August, 1990.

SELECT page q
FROM cube P
WHERE q.profile.dimension.category = document
and q.profile.dimension.class = rules
and q.profile.dimension.topic
IN {leave.casual.*, travel..abroad.*}
and q.profile.dimension.time BETWEEN {1990.08.*, now}
ORDER BY dimension.topic, dimension.time

The query produces a resultant page from a page cube $P$ containing links to all the pages containing rules on casual leave, or on travel.abroad,
or on any subtopics, produced between August 1990 and now. The value of the special variable now is defined by the arbiter with the current time stamp. The values included in the IN list are generally values of dimensional attributes of cube P. It is a shorter form of expressing ORs of attribute-value based predicates. The range of values in BETWEEN clause is a more general expression of IN list.

4. Find all documents from a page cube P containing rules on special casual leave and include them in the context of the workflow w.

( SELECT page q
    FROM cube P
    WHERE q.profile.dimension.category = context
    and q.profile.dimension.dpw = w)
PLUG ( SELECT p
    FROM P
    WHERE p.profile.dimension.category = document
    and p.profile.dimension.class = rule
    and p.profile.dimension.topic = leave.casual.special )

5. Find all parts of the dpw w from a page cube P where the page with pageId "2000.08.12.13.25.31" is cited.

SELECT page p
FROM cube P, P.graph.category g
WHERE p.category = part
and g.dpw = w
3.3.4 PSQL Grammar

In this section the core of the Context-Free Grammar of PSQL in BNF notations is given. Since PSQL is a an extension of SQL, therefore, some of the productions are from standard SQL, some are from Active Rule [41]. Some productions are similar to DSQL [35] and Lorel[31]. Others are specific to PSQL.

\[
\begin{align*}
\text{<query-exp>} & ::= \text{<query_term> <query-exp> | <query_term>} \\
\text{<query_term>} & ::= \text{<rule_spec> | <query_spec>} \\
\text{<rule_spec>} & ::= \text{WHEN <event> [IF <cond_exp> THEN] <query_spec>} \\
\text{<event>} & ::= \text{request | response} \\
\text{<query_spec>} & ::= \text{SELECT [ALL | DISTINCT] <query_body>} \\
& \quad [\{\text{PLUG | UNPLUG | UNION | INTERSECTION | DIFFERENCE}\} <query_spec>] \\
\text{<if_clause>} & ::= \text{IF <cond_exp> THEN \{<if_clause>|<query_spec>\}} \\
\text{<query_body>} & ::= \text{<from_clause> [\<where_clause|>]} [\<order_clause|>] \\
\text{<order_clause>} & ::= \text{ORDER BY (path_list)} \\
\text{<from_clause>} & ::= \text{FROM <path_exp> <identifier>]} \\
& \quad [, <path_exp> <identifier>] \\
\text{<where_clause>} & ::= \text{WHERE <cond_exp>} \\
\text{<cond_exp>} & ::= \text{<cond_term> | <cond_exp> OR <cond_term>} \\
\text{<cond_term>} & ::= \text{<cond_factor> | <cond_term> AND <cond_factor>} \\
\text{<cond_factor>} & ::= \text{[NOT] <cond_test>}
\end{align*}
\]
3.4 Closure

Page Cube is a closed model. Simply put, this means that input to a query are pages belonging to PC and the result-set, that is, the output of a query, is also a page having hyperlinks to the pages satisfying the query. Closure is prominent in a relational database model, as tables are both input to queries and output from queries. In addition using the QBE query language, queries are formulated also using tables. Similarly, in page cube, queries can also be pages. The advantages of a closed model are discussed in [35]
3.5 Discussion

In this Chapter a model, called Page Cube, for storage and retrieval of documents in an office environment is presented. The model is a multi-dimensional model. Multi-dimensional modelling is a new paradigm of modelling data. Data Cube model of OLAP is the most common application of multi-dimensional modelling that influenced the data warehouse architecture. In data cube, numeric measures, like sales, inventory, population etc., are the main subject of the analysis. Unlike ER modelling, which describes entities and relationships, dimension modelling deals with numeric measures and the dimensions. It is for the first time, in the Page Cube model, dimension modelling is extended in modelling office documents. The Page Cube deals with dimensions and pages and their inter-connectivity. The genesis of Page Cube model is the formation of page graphs with temporal labelling scheme of the edges. Query languages like Relational Algebra and SQL are very rich in constructs. But for querying pages from a Page Cube only a small and simple set of constructs are necessary. Moreover, some constructs are specific to Page Cube. Therefore, two query languages, in the line of RA and SQL, namely PA and PSQL are proposed to query Page Cubes. The model can be extended by incorporating document to document linking by corresponding kinds of edges if such a need arises in a particular office.