1. Introduction to term project
   1. **What did I do:** Design and query relational database, means to import arbitrary user-defined xml documents into the relational database to then query.
   2. Explain why I deviated from a true xml query language and explain misunderstanding with what it actually means to query xml.
   3. **Motivation**:  **(Why?)** 
      1. Gain a unique perspective how database vendors went about creating relational database. Understand difficulties involved, from many facets such as how to store data in a volatile fashion, how to represent as a memory structure to query faster, how to parse queries, learning best way to implement efficient algorithms in both selecting, joining, filtering, aggregating, and sorting.
      2. Learn through research what the concept of XML, what it means to the world today, what it is used for, if it excels at what it does, its evolution from historical understanding to current understanding of its function, and of course how to manage, exchange, handle xml.
      3. Understand how xml fits in with current database structures, most specifically relational databases, and how I might perhaps make my relational database pseudo “xml-enabled” in that it will be able to extract arbitrary user-provided xml files and store it into a relational view such that it can be queried relationally (SQL).
      4. AND of course learn more about WPF, have the privilege of utilizing state of the art technologies (Visual Studio 2010, .Net 4.0, Windows Presentation Foundation along with its controls), utilize an XML API provided with .Net in handling my xml data-source as well as the DataSet/DataTable object.
   4. Transition – “But first let me introduce the concept of XML from both its historical view to its now current view. What it is used for, its structure, format, why has it gained the status it has, how to manage and query it. “
2. XML
   1. **Intro to XML: Extensible Markup Language (XML).**
      1. Origins from Standard Generalized Markup Language (SGML) in that it was first used for structuring large documents.
      2. De-facto standard in data exchange that unlike HTML and SGML, it can represent database data. (Not historically understood as a database concept, but gradually evolved into one over time for its usefulness in structuring data arbitrarily)
   2. **Motivation**
      1. **Markup** **Language** means something that doesn’t appear to the end user (traditionally as blue ink revision instructions by editors). Imposes a structuring to the data. Include typesetting instructions (Font, Bold) but evolved into a more functional representation. <Header, Text, Title…> Now each of these parts can be formatted uniformly. **Picture**. Briefly explain tags and how they are contained with < > brackets.
      2. Set of tags not established and enforced by XML. Structuring of tags/prescribed predetermined allowable tags can be established at will depending on situation or business/application environment.
      3. Useful for data exchange – all data can be exchanged using one document that can evolve over time. Previously purchase orders needed to be output from source application, transcribed on paper manually, sent to destination site, interpreted, and manually input into the destination application/database. Now data can be directly sent between applications automatically (mechanically) in the form of xml that can be easily published and then shredded (imported). (Process greatly sped up and facilitated because of xml).
      4. Flexible – New tags (LastPurchasedDate) may be added at will from source and safely ignored at target, meaning data can evolve over time with minimal overhead and complications. Managing application won’t break as easily.
      5. Deeply complex nested structuring . Same named tags can be repeated in a level allowing for easy representation of multiple valued attributes. (Such as list of products bought by a customer).
      6. **Major Benefits/Downsides vs relational database**: Deeply complex nested structuring can be built at will. Self-documenting (No schema needed to understand). All information can be stored in one document vs multiple relations and keys needed for relational data, thus easy to exchange data. Flexibily - updated without worry about managing application. **Downside**: Redundant data – If different people order the same product, the long product description would be duplicated each time. Inefficiency introduced because of redundant data and the need for using and processing <TAGS> (Thus containing data size is bloated compared to relational database data).
      7. XML Format is widely accepted, and thus many industry-standard applications have been built to facilitate managing and exchanging xml.
   3. **XML Structure/Terminology**
      1. **XML Document, Single Root encompassing all other elements.**
      2. **Element** - fundamental component of XML that contains data within its tags. Within matching starting and ending tags, or optionally with < />, nesting elements within elements as long as you want. Must be **properly nested**. As mentioned before, complex nesting means 1 document and can avoid cumbersome joins! **Attributes** – contain only text values within element tags. Distinction not so apparent between attributes and single valued sub-elements.
      3. **Namespace –** To act as an acceptable exchange facility, must be able to keep tags unique across agencies/companies/businesses. Web URL can be used to uniquely identify a tag. <bank xmlns:FB= “http://www.Firstbank.com”> <FB:tagName> Data </FB:tagName>. **Default Namespace** – No need to explicitly specify tag namespace prefix for default namespace.
   4. **DTD/XML Schema**
      1. Downside to current explanation of XML structuring is an apparent inability to force structuring if need be to ensure both source and destination sites sync in their understanding of format (especially in case of automatic programmatic exchange of data between applications). Furthermore, inability to represent types, enforce allowed ranges on values, number of children sub-elements allowed, impose required ordering on sub-elements. Will talk about two schemas introduced in order to impose more of a structure to XML—both the original schema in Document Type Definition (DTD) and newer XML Schema.
      2. **Document Type Definition** (DTD) – optional part of xml document, first schema definition language included as part of xml standard. **(DTD is basically a list of rules that the included XML Document will adhere to in presenting its data.)** Fails to introduce concept of typed values within elements. Exceptional at ability to enforce structuring of elements, element names, attributes that must appear, ID’s of attributes (meaning ID can appear only once in an element in an xml document), REFID’s (references of ID’s that must exist currently in document), ordering of elements. | (or), + (one or more), \* (zero or more), ? (zero or one optional element), #PCDATA (parsed character data), **DTD PICTURE**.  **DOWNSIDES:** Lack of data typing, unable to easily specify that data can be unordered (not as flexible), can’t explicitly link a particular IDREF with a particular ID.
      3. **XML Schema** –Currently accepted schema structuring for XML built and improved off of DTD. **XML SCHEMA PICTURE**.
         1. Allowable types: string, integer, decimal, date, Boolean. Complex types included sequences of sub-elements. Also enforce range on values. (Contrasted to just the normal elements containing data, which have just simple types string integer…). **User defined types.**
         2. Enforce uniqueness, primary and foreign key relationships, allowable ranges that an element can appear (minOccurs = “0”, maxOccurs = “Unbounded”).
         3. Last but not least it uses xml structure itself, so that it can easily be transported along with its corresponding XML Document.
   5. **Querying – Xpath/XQuery.** 
      1. As output of a relational query is a relation, output of xml query is another xml document.
      2. Important to understand that the structure of XML is inevitably a tree model – one document root, multiple children elements, each having children, siblings, attributes, then of course the data values found between tags and in attributes. Because of this complexity in the data, any xml query language will inevitably be complex and need to be able to easily traverse the tree structure, break it apart, transform it, etc. while providing an easy front-end language for users to manipulate any arbitrary xml of their choice.
      3. **XPath** – Language for **path expressions** allowing easy selection of current xml data through means of directory type of structure/syntax with slash ‘/’. Attributes can be accessed with @. Acts as a tool that XQuery and other xml query languages can use to navigate files more easily. Also contains crude form of selection predicates, aggregate functions (**count)** , selection of attribute values**. Function id(“1234554”)** can be applied on sets of references or strings containing multiple references to ID’s. It then returns those exact nodes (and their children) that contain that exact ID Value (in one of its ID attributes). The | operator allows results to be unioned. ‘//’ allows you to search all descendent elements for a particular child element name. ‘..’ does the opposite but for parents. **Doc(.. )** allows you to search a specific xml document other than the default one you’re in.
      4. **XQUERY**
         1. Queries xml documents returning other xml documents.
         2. Modeled after SQL queries which is impressive given its need to search complex nested structuring of xml data, manipulate and transform data, and output the data in a flexible manner.
         3. Surprisingly powerful including a vast range of transformational techniques including filtering, sorting, aggregating, nested queries, joins, temporary variables, user defined functions, if-then-else commands, **some** or **every** keyword (existential/universal quantifiers),
         4. **FLWOR** (“flower”) **For, let, where, order by, return** – explain them from first example. For clause like from clause in that it searches data and returns a subset of the data that can then be transformed (enforces scoping). Let clause is optional and allows one to rename expression paths to variable names that can be easily referred to later in a query. Where filters, return returns an xml document. Explain { }.
         5. Only needed are **for** and **return**. Can equivalently use xpath expressions with their selection predicates instead of using **where** clause.
         6. **Joining** – Similar to SQL style joins: Cartesian product, filtered using where or even xpath expressions inline.
         7. **Comparison operators** – Interesting concept. If multiple values are returned >, >=, <, <= will return true if at least one of those values satisfies the comparison.
         8. **Nested Queries** – Nesting takes place in **return** clause.
         9. **Aggregate functions, distinct values, sorting** – Sum(), Count(), Distinct-values(), aggregate functions must be done in **return** clause. Sorting of results done by use of **order by** keyword. Sorting can be done on multiple levels within nested queries. Once in the outer layer, and a second time within a nested query.
         10. Can define user designed functions
   6. API in terms of XML
      1. Two standard models for programmatic manipulation of xml data.
      2. DOM – Document Object Model. Treats xml content as a tree that can programmatically be traversed by calling functions such as **GetParentNode()**, GetFirstChild(), GetNextSibling(), GetElementsByTagName(name) (to get sub elements), GetAttributeValue(name), GetDate() – returns text value contained within an element.
      3. Simple API for XML (SAX) – API built on notion of event handlers. SAX traverses structure from beginning to end firing events such as start-tag /end-tag found for an element or attribute, text value is found, document tag is ended. Developer builds handlers to handle the event and populates his/her own desired data structure of the data.
   7. **Storage of XML Data**
      1. Import to be able to store xml data effectively and efficiently. 3 possible ways:
      2. Way 1: Store documents in a folder or file system held in their original flat file structure.xml. Bereft of data isolation, atomicity, concurrent access, and security. Easy to manage though.
      3. Way 2: XML Database – works only on xml, using xml data as its basic data model. Complex undertaking… better to use currently existing relational databases and build an abstraction layer on top
      4. Way 3: Relational databases: XML-Enabled databases.
         1. **Store as strings** – clob or blob. Downsides are no inherent understanding of schema of stored strings, not possible to query data directly, not even possible to implement simple selections such as finding all elements of a certain label (Account, Customer). No way to index values in the xml string.
         2. **Tree representation** – Data modeled as a tree with many relations. Each element and attribute in xml data is given a unique ID. *Nodes(id, type, label, value) child(child\_id, parent\_id)****.*** Type: is element or attribute. **Advantage**: Can be queried relationally once structure is understood by user. **Disadvantage:** cumbersome to manage all the relations and join them all (lots of joins introduced).
         3. **Map to relations** - All attributes are stored as attribute in a relation. Relations are created for each element name. Subelements are created as an attribute in that relation if it’s a single valued subelement (no nested elements within). Otherwise complex elements are given a new relation, given an id, and each sub element is connected to it via a parent\_id.
         4. **Native storage within a relational database/ recently defined SQL/XML standard –** recent advent of new data type within sql is type **xml**. Allows entire xml documents to be stored in a different tuple that can then be queried using xquery language within sql itself. Furthermore, select syntax has been enhanced to be able to return xml results instead of relational results. **PICTURE**.
3. **DEMO**
4. **Implementation**
5. **Conclusion**
   1. Great learning experience. Enjoyed learning about xml. Enjoyed implementing relational database and coming up with a way to make the datatbase xml-enabled, though its xml support was very limited.
   2. Future work
      1. Would enjoy enhancing database with a parse tree, more robust forms of parsing, tokenizing, etc.
      2. Would like to implement the concept of nullable values, unique constraints, foreign keys, primary keys, fully supporting theta joins (with or’s allowed also), outer joins, nested queries, existential and universal quantification, filtering mechanisms on grouped/aggregated data.
      3. Of course this was a fun little project and I was surprised and the quantity of functionality I could introduce within the system already.
      4. Would further like to make the database more xml-enabled and implement some of the algorithms such as **Mapping to Relations**. Though this might be a formidable task, it would be interesting to see how mapping to relations would truly work in a real world scenario first-hand instead of just a concept.