



Database Systems

Lecture 1: Course Overview

Dr. Momtazi
momtazi@aut.ac.ir

Provided by: Ali Moradzade 9831058

based on the slides of the course book



Outline

- **Administrative Information**
- Introduction to the Course
- Overview of the Semester



Course Home Page

- Administrative information
- Slides
- Exercises



Assessment

- Regular attendance in the class
 - More than 3/16 absences will be reported
- Final exam (50%)
- Midterm exam (20%)
- Exercises (20%)
- Final project (10%)



Teaching

- Both theoretical and practical concepts in main sessions
- More practical sessions by teacher assistant
 - Goals:
 - Solving exercises
 - Teaching MySQL



Contact

- Email: momtazi@aut.ac.ir
- Phone: 021-64542737



Text Book

Database System Concepts

by Abraham Silberschatz

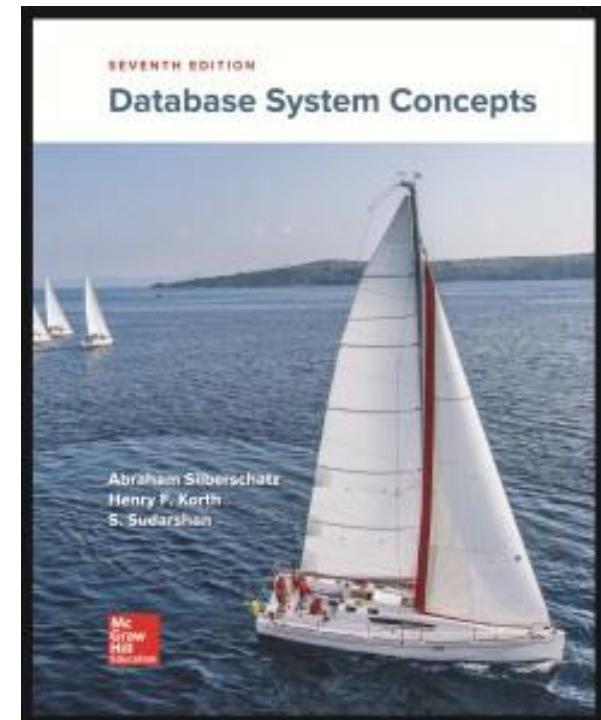
Henry F. Korth

S. Sudarshan

7th EDITION

Publisher: McGraw-Hill

2019





Rules of the Game

- In case you don't understand something:
 - Ask!!!
 - Ask!!!
 - Ask!!!



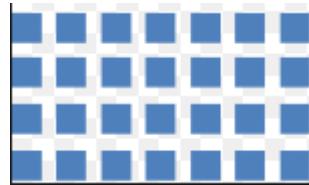
Outline

- Administrative Information
- **Introduction to the Course**
- Overview of the Semester

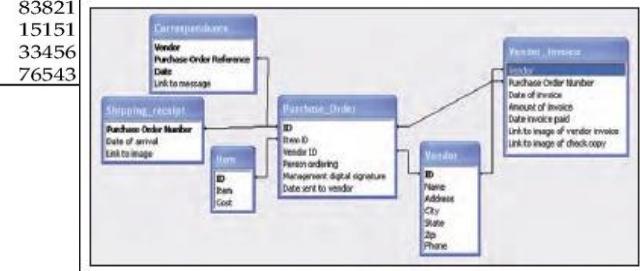


Data

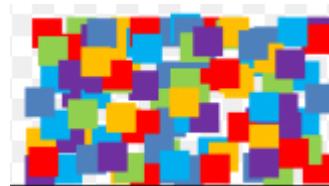
■ Structured



<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000



■ Unstructured



Techniques such as **data mining**, **Natural Language Processing(NLP)**, and **text analytics** provide different means to interpret this information. Common techniques for structuring text usually involve manual tagging with metatags further **text mining**-based structuring. **Unstructured Information Management Architecture (UIMA)** provides a framework for extracting meaning and creating structured data about the information.^[5]

Software that creates machine-processable structure exploits the linguistic, auditory, and visual structure in communication.^[6] Algorithms can infer this inherent structure from text, for instance, by examining word *m* small- and large-scale patterns. Unstructured information can then be enriched and tagged to address amb then used to facilitate search and discovery. Examples of "unstructured data" may include books, journals, c audio, video, analog data, images, files, and unstructured text such as the body of an e-mail message. Web the main content being conveyed does not have a defined structure, it generally comes packaged in object themselves have structure and are thus a mix of structured and unstructured data, but collectively this is st example, an HTML web page is tagged, but HTML mark-up typically serves solely for rendering. It does not c elements in ways that support automated processing of the information content of the page. XHTML tagging elements, although it typically does not capture or convey the semantic meaning of tagged terms.

Since unstructured data commonly occurs in [electronic documents](#), the use of a [content or document management system](#) is often preferred over data transfer and manipulation from within the documents. Document management systems are designed to manage entire documents and to convey structure onto [document collections](#).

Search engines have become popular tools for indexing and searching through such data, especially text.



Structured vs. Unstructured Data

Albert Einstein

From Wikipedia, the free encyclopedia

Albert Einstein (14 March 1879 – 18 April 1955) was a German-American theoretical physicist who developed the theory of general relativity, effecting a revolution in physics. For his achievements, Einstein is often regarded as the father of *modern physics* and one of the most prolific *geniuses* in human history.^[1] He received the 1921 Nobel Prize in Physics "for his services to theoretical physics, and especially for his discovery of the law of the photoelectric effect".^[2] This latter was pivotal in establishing quantum theory within physics.

near the beginning of his career, Einstein proposed that the mechanics and motion of particles was in contradiction with the laws of classical mechanics with the laws of the electromagnetic field. This led to the development of his special theory of relativity. He realized, however, that the principle of relativity could also be extended to gravitational fields, and with his field equations of 1915, he published a paper on the general theory of relativity, which led to the development of general relativity. He also investigated the thermal properties of light which led to the foundation of the photon theory of light. In 1917, Einstein applied the general theory of relativity to model the structure of the universe as a whole.^[3]

the most exciting period of his life when *relativity* came to power in 1915, and did not go back to Germany, where he had been a professor at the Berlin Academy of Sciences. He settled in the U.S., becoming a citizen in 1940 (on the eve of World War II), he helped Albert Einstein Institute (AEI) founded in Germany right before developing a atomic weapon, and recommended that the U.S. begin similar research; this eventually led to what would become the *Manhattan Project*. Einstein may be known for defending the *atheist* forces, but largely deserved being the new discoverer of *atheist* forces as a scientist. Later, together with Niels Bohr, Einstein signed the *Princeton-Einstein Manifesto*, which highlighted the danger of nuclear weapons.

Einstein published more than 600 scientific papers along with over 100 non-scientific works.^[4] His great intelligence and originality have made the word 'genius' synonymous with genius.^[5]

Contents

- Biography
- Early life and education
- Work
- Personal life
- Death and legacy
- Honors and awards
- References
- External links
- Further reading
- See also
- Related topics
- References
- External links
- Further reading
- See also

Biography

1. Early life and education

2. Work

3. Personal life

4. Death and legacy

5. Honors and awards

6. References

7. External links

8. Further reading

9. See also

10. Related topics

11. References

12. External links

13. Further reading

14. See also

15. Related topics

16. References

17. External links

18. Further reading

19. See also

20. Related topics

21. References

22. External links

23. Further reading

24. See also

25. Related topics

26. References

27. External links

28. Further reading

29. See also

30. Related topics

31. References

32. External links

33. Further reading

34. See also

35. Related topics

36. References

37. External links

38. Further reading

39. See also

40. Related topics

41. References

42. External links

43. Further reading

44. See also

45. Related topics

46. References

47. External links

48. Further reading

49. See also

50. Related topics

51. References

52. External links

53. Further reading

54. See also

55. Related topics

56. References

57. External links

58. Further reading

59. See also

60. Related topics

61. References

62. External links

63. Further reading

64. See also

65. Related topics

66. References

67. External links

68. Further reading

69. See also

70. Related topics

71. References

72. External links

73. Further reading

74. See also

75. Related topics

76. References

77. External links

78. Further reading

79. See also

80. Related topics

81. References

82. External links

83. Further reading

84. See also

85. Related topics

86. References

87. External links

88. Further reading

89. See also

90. Related topics

91. References

92. External links

93. Further reading

94. See also

95. Related topics

96. References

97. External links

98. Further reading

99. See also

100. Related topics

101. References

102. External links

103. Further reading

104. See also

105. Related topics

106. References

107. External links

108. Further reading

109. See also

110. Related topics

111. References

112. External links

113. Further reading

114. See also

115. Related topics

116. References

117. External links

118. Further reading

119. See also

120. Related topics

121. References

122. External links

123. Further reading

124. See also

125. Related topics

126. References

127. External links

128. Further reading

129. See also

130. Related topics

131. References

132. External links

133. Further reading

134. See also

135. Related topics

136. References

137. External links

138. Further reading

139. See also

140. Related topics

141. References

142. External links

143. Further reading

144. See also

145. Related topics

146. References

147. External links

148. Further reading

149. See also

150. Related topics

151. References

152. External links

153. Further reading

154. See also

155. Related topics

156. References

157. External links

158. Further reading

159. See also

160. Related topics

161. References

162. External links

163. Further reading

164. See also

165. Related topics

166. References

167. External links

168. Further reading

169. See also

170. Related topics

171. References

172. External links

173. Further reading

174. See also

175. Related topics

176. References

177. External links

178. Further reading

179. See also

180. Related topics

181. References

182. External links

183. Further reading

184. See also

185. Related topics

186. References

187. External links

188. Further reading

189. See also

190. Related topics

191. References

192. External links

193. Further reading

194. See also

195. Related topics

196. References

197. External links

198. Further reading

199. See also

200. Related topics

201. References

202. External links

203. Further reading

204. See also

205. Related topics

206. References

207. External links

208. Further reading

209. See also

210. Related topics

211. References

212. External links

213. Further reading

214. See also

215. Related topics

216. References

217. External links

218. Further reading

219. See also

220. Related topics

221. References

222. External links

223. Further reading

224. See also

225. Related topics

226. References

227. External links

228. Further reading

229. See also

230. Related topics

231. References

232. External links

233. Further reading

234. See also

235. Related topics

236. References

237. External links

238. Further reading

239. See also

240. Related topics

241. References

242. External links

243. Further reading

244. See also

245. Related topics

246. References

247. External links

248. Further reading

249. See also

250. Related topics

251. References

252. External links

253. Further reading

254. See also

255. Related topics

256. References

257. External links

258. Further reading

259. See also

260. Related topics

261. References

262. External links

263. Further reading

264. See also

265. Related topics

266. References

267. External links

268. Further reading

269. See also

270. Related topics

271. References

272. External links

273. Further reading

274. See also

275. Related topics

276. References

277. External links

278. Further reading

279. See also

280. Related topics

281. References

282. External links

283. Further reading

284. See also

285. Related topics

286. References

287. External links

288. Further reading

289. See also

290. Related topics

291. References

292. External links

293. Further reading

294. See also

295. Related topics

296. References

297. External links

298. Further reading

299. See also

300. Related topics

301. References

302. External links

303. Further reading

304. See also

305. Related topics

306. References

307. External links

308. Further reading

309. See also

310. Related topics

311. References

312. External links

313. Further reading

314. See also

315. Related topics

316. References

317. External links

318. Further reading

319. See also

320. Related topics

321. References

322. External links

323. Further reading

324. See also

325. Related topics

326. References

327. External links

328. Further reading

329. See also

330. Related topics

331. References

332. External links

333. Further reading

334. See also

335. Related topics

336. References

337. External links

338. Further reading

339. See also

340. Related topics

341. References

342. External links

343. Further reading

344. See also

345. Related topics

346. References

347. External links

348. Further reading

349. See also

350. Related topics

351. References

352. External links

353. Further reading

354. See also

355. Related topics

356. References

357. External links

358. Further reading

359. See also

360. Related topics

361. References

362. External links

363. Further reading

364. See also

365. Related topics

366. References

367. External links

368. Further reading

369. See also

370. Related topics

371. References

372. External links

373. Further reading

374. See also

375. Related topics

376. References

377. External links

378. Further reading

379. See also

380. Related topics

381. References

382. External links

383. Further reading

384. See also

385. Related topics

386. References

387. External links

388. Further reading

389. See also

390. Related topics

391. References

392. External links

393. Further reading

394. See also

395. Related topics

396. References

397. External links

398. Further reading

399. See also

400. Related topics

401. References

402. External links

403. Further reading

404. See also

405. Related topics

406. References

407. External links

408. Further reading

409. See also

410. Related topics

411. References

412. External links

413. Further reading

414. See also

415. Related topics

416. References

417. External links

418. Further reading

419. See also

420. Related topics

421. References

422. External links

423. Further reading

424. See also

425. Related topics

426. References

427. External links

428. Further reading

429. See also

430. Related topics

431. References

432. External links

433. Further reading

434. See also

435. Related topics

436. References

437. External links

438. Further reading

439. See also

440. Related topics

441. References

442. External links

443. Further reading

444. See also

445. Related topics

446. References

447. External links

448. Further reading

449. See also

450. Related topics

451. References

452. External links

453. Further reading

454. See also

455. Related topics

456. References

457. External links

458. Further reading

459. See also

460. Related topics

461. References

462. External links

463. Further reading

464. See also

465. Related topics

466. References

467. External links

468. Further reading

469. See also

470. Related topics

471. References

472. External links

473. Further reading

474. See also

475. Related topics

476. References

477. External links

478. Further reading

479. See also

480. Related topics

481. References

482. External links

483. Further reading

484. See also

485. Related topics

486. References

487. External links

488. Further reading

489. See also

490. Related topics

491. References

492. External links

493. Further reading

494. See also

495. Related topics

496. References

497. External links

498. Further reading

499. See also

500. Related topics

501. References

502. External links

503. Further reading

504. See also

505. Related topics

506. References

507. External links

508. Further reading

509. See also

510. Related topics

511. References

512. External links

513. Further reading

514. See also

515. Related topics

516. References

517. External links

518. Further reading

519. See also

520. Related topics

521. References

522. External links

523. Further reading

524. See also

525. Related topics

526. References

527. External links

528. Further reading

529. See also

530. Related topics

531. References

532. External links

533. Further reading

534. See also

535. Related topics

536. References

537. External links

538. Further reading

539. See also

540. Related topics

541. References

542. External links

543. Further reading

544. See also

545. Related topics

546. References

547. External links

548. Further reading

549. See also

550. Related topics

551. References

552. External links

553. Further reading

554. See also

555. Related topics

556. References

557. External links

558. Further reading

559. See also

560. Related topics

561. References

562. External links

563. Further reading

564. See also

565. Related topics

566. References

567. External links

568. Further reading

569. See also

570. Related topics

571. References

572. External links

573. Further reading

574. See also

575. Related topics

576. References

577. External links

578. Further reading

579. See also

580. Related topics

581. References

582. External links

583. Further reading

584. See also

585. Related topics

586. References

587. External links

588. Further reading

589. See also

590. Related topics

591. References

592. External links

593. Further reading

594. See also

595. Related topics

596. References

597. External links

598. Further reading

599. See also

600. Related topics

601. References

602. External links

603. Further reading

604. See also

605. Related topics

606. References

607. External links

608. Further reading

609. See also

610. Related topics

611. References

612. External links

613. Further reading

614. See also

615. Related topics

616. References

617. External links

618. Further reading

619. See also

620. Related topics

621. References

622. External links

623. Further reading

624. See also

625. Related topics

626. References

627. External links

628. Further reading

629. See also

630. Related topics

631. References

632. External links

633. Further reading

634. See also

635. Related topics

636. References

637. External links

638. Further reading

639. See also

640. Related topics

641. References

642. External links

643. Further reading

644. See also

645. Related topics

646. References

647. External links

648. Further reading

649. See also

650. Related topics

651. References

652. External links

653. Further reading

654. See also

655. Related topics

656. References

657. External links

658. Further reading

659. See also

660. Related topics

661. References

662. External links

663. Further reading

664. See also

665. Related topics

666. References

667. External links

668. Further reading

669. See also

670. Related topics

671. References

672. External links

673. Further reading

674. See also

675. Related topics

676. References

677. External links

678. Further reading

679. See also

680. Related topics

681. References

682. External links

683. Further reading

684. See also

685. Related topics

686. References

687. External links

688. Further reading

689. See also

690. Related topics

691. References

692. External links

693. Further reading

694. See also

695. Related topics

696. References

697. External links

698. Further reading

699. See also

700. Related topics

701. References

702. External links

703. Further reading

704. See also

705. Related topics

706. References

707. External links

708. Further reading

709. See also

710. Related topics

711. References

712. External links

713. Further reading

714. See also

715. Related topics

716. References

717. External links

718. Further reading

719. See also

720. Related topics

721. References

722. External links

723. Further reading

724. See also

725. Related topics

726. References

727. External links

728. Further reading

729. See also

730. Related topics

731. References

732. External links

733. Further reading

734. See also

735. Related topics

736. References

737. External links

738. Further reading

739. See also

740. Related topics

741. References

742. External links

743. Further reading

744. See also

745. Related topics

746. References

747. External links

748. Further reading

749. See also

750. Related topics

751. References

752. External links

753. Further reading

754. See also

755. Related topics

756. References

757. External links

758. Further reading

759. See also

760. Related topics

761. References

762. External links

763. Further reading

764. See also

765. Related topics

766. References

767. External links

768. Further reading

769. See also

770. Related topics

771. References

772. External links

773. Further reading

774. See also

775. Related topics

776. References

777. External links

778. Further reading

779. See also

780. Related topics

781. References

782. External links

783. Further reading

784. See also

785. Related topics

786. References

787. External links

788. Further reading

789. See also

790. Related topics

791. References

792. External links

793. Further reading

794. See also

795. Related topics

796. References

797. External links

798. Further reading

799. See also

800. Related topics

801. References

802. External links

803. Further reading

804. See also

805. Related topics

806. References

807. External links

808. Further reading

809. See also

810. Related topics

811. References

812. External links

813. Further reading

814. See also

815. Related topics

816. References

817. External links

818. Further reading

819. See also

820. Related topics

821. References

822. External links

823. Further reading

824. See also

825. Related topics

826. References

827. External links

828. Further reading

829. See also

830. Related topics

831. References

832. External links

833. Further reading

834. See also

835. Related topics

836. References

837. External links

838. Further reading

839. See also

840. Related topics

841. References

842. External links

843. Further reading

844. See also

845. Related topics

846. References

847. External links

848. Further reading

849. See also

850. Related topics

851. References

852. External links

853. Further reading

854. See also

855. Related topics

856. References

857. External links

858. Further reading

859. See also

860. Related topics

861. References

862. External links

863. Further reading

864. See also

865. Related topics

866. References

867. External links

868. Further reading

869. See also

870. Related topics

871. References

872. External links

873. Further reading

874. See also

875. Related topics

876. References

877. External links

878. Further reading

879. See also

880. Related topics

881. References

882. External links

883. Further reading

884. See also

885. Related topics

886. References

887. External links

888. Further reading

889. See also

890. Related topics

891. References

892. External links

893. Further reading

894. See also

895. Related topics

896. References

897. External links

898. Further reading

899. See also

900. Related topics

901. References

902. External links

903. Further reading

904. See also

905. Related topics

906. References

907. External links

908. Further reading

909. See also

910. Related topics

911. References

912. External links

913. Further reading

914. See also

915. Related topics

916. References

917. External links

918. Further reading

919. See also

920. Related topics

921. References

922. External links

923. Further reading

924. See also

925. Related topics

926. References

927. External links

928. Further reading

929. See also

930. Related topics

931. References

932. External links

933. Further reading

934. See also

935. Related topics

936. References

937. External links

938. Further reading

939. See also

940. Related topics

941. References

942. External links

943. Further reading

944. See also

945. Related topics

946. References

947. External links

948. Further reading

949. See also

950. Related topics

951. References

952. External links

953. Further reading

954. See also

955. Related topics

956. References

957. External links

958. Further reading

959. See also

960. Related topics

961. References

962. External links

963. Further reading

964. See also

965. Related topics

966. References

967. External links

968. Further reading

969. See also

970. Related topics

971. References

972. External links

973. Further reading

974. See also

975. Related topics

976. References

977. External links

978. Further reading

979. See also

980. Related topics

981. References

982. External links

983. Further reading

984. See also

985. Related topics

986. References

987. External links

988. Further reading

989. See also

990. Related topics

991. References

992. External links

993. Further reading

994. See also

995. Related topics

996. References

997. External links

998. Further reading

999. See also

1000. Related topics

1001. References

1002. External links

1003. Further reading

1004. See also

1005. Related topics

1006. References

1007. External links

1008. Further reading

1009. See also

1010. Related topics

1011. References

1012. External links

1013. Further reading

1014. See also

1015. Related topics

1016. References

1017. External links

1018. Further reading

1019. See also

1020. Related topics

1021. References

1022. External links

1023. Further reading

1024. See also

1025. Related topics

1026. References

1027. External links

1028. Further reading

1029. See also

1030. Related topics

1031. References

1032. External links

1033. Further reading

1034. See also

1035. Related topics

1036. References

1037. External links

1038. Further reading

1039. See also

1040. Related topics

1041. References

1042. External links

1043. Further reading

1044. See also

1045. Related topics

1046. References

1047. External links

1048. Further reading

1049. See also

1050. Related topics

1051. References

1052. External links

1053. Further reading

1054. See also

1055. Related topics

1056. References

1057. External links

1058. Further reading

105



Data

■ Storage



■ Retrieval





The Need for Databases

- The Internet revolution of the late 1990s sharply increased direct user access to databases.
- Converting many of phone interfaces into Web interfaces
- Making a variety of services and information available online.
 - Accessing an online bookstore and browse a book or music collection
 - Entering an order online
 - Accessing a bank Web site and retrieving bank balance and transaction information
 - Accessing a Web site and browsing its advertisement



The Need for Databases

- Database system vendors are among the largest software companies in the world





Database Management System (DBMS)

- DBMS contains information about a particular enterprise
 - Collection of interrelated data
 - Set of programs to access the data
 - An environment that is both *convenient* and *efficient* to use
- Databases can be very large.
- Databases touch all aspects of our lives



Database Applications

- Banking: transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Online retailers: order tracking, customized recommendations
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions



University Database Example

- Application program examples
 - Add new students, instructors, and courses
 - Register students for courses, and generate class rosters
 - Assign grades to students, compute grade point averages (GPA) and generate transcripts
- In the early days, database applications were built directly on top of file systems



Drawbacks of using file systems to store data

- Data redundancy and inconsistency
 - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
 - Need to write a new program to carry out each new task
- Data isolation
 - Multiple files and formats
- Integrity problems
 - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
 - Hard to add new constraints or change existing ones



Drawbacks of using file systems to store data

- Atomicity of updates
 - Failures may leave database in an inconsistent state with partial updates carried out
 - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
 - Concurrent access needed for performance
 - Uncontrolled concurrent accesses can lead to inconsistencies
 - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
 - Hard to provide user access to some, but not all, data



Outline

- Administrative Information
- Introduction to the Course
- **Overview of the Semester**



View of Data

- A database system is a collection of interrelated data and a set of programs
- Allow users to access and modify these data
- Major purposes:
 - Providing users with an abstract view of the data
 - Hiding certain details of how the data are stored and maintained.



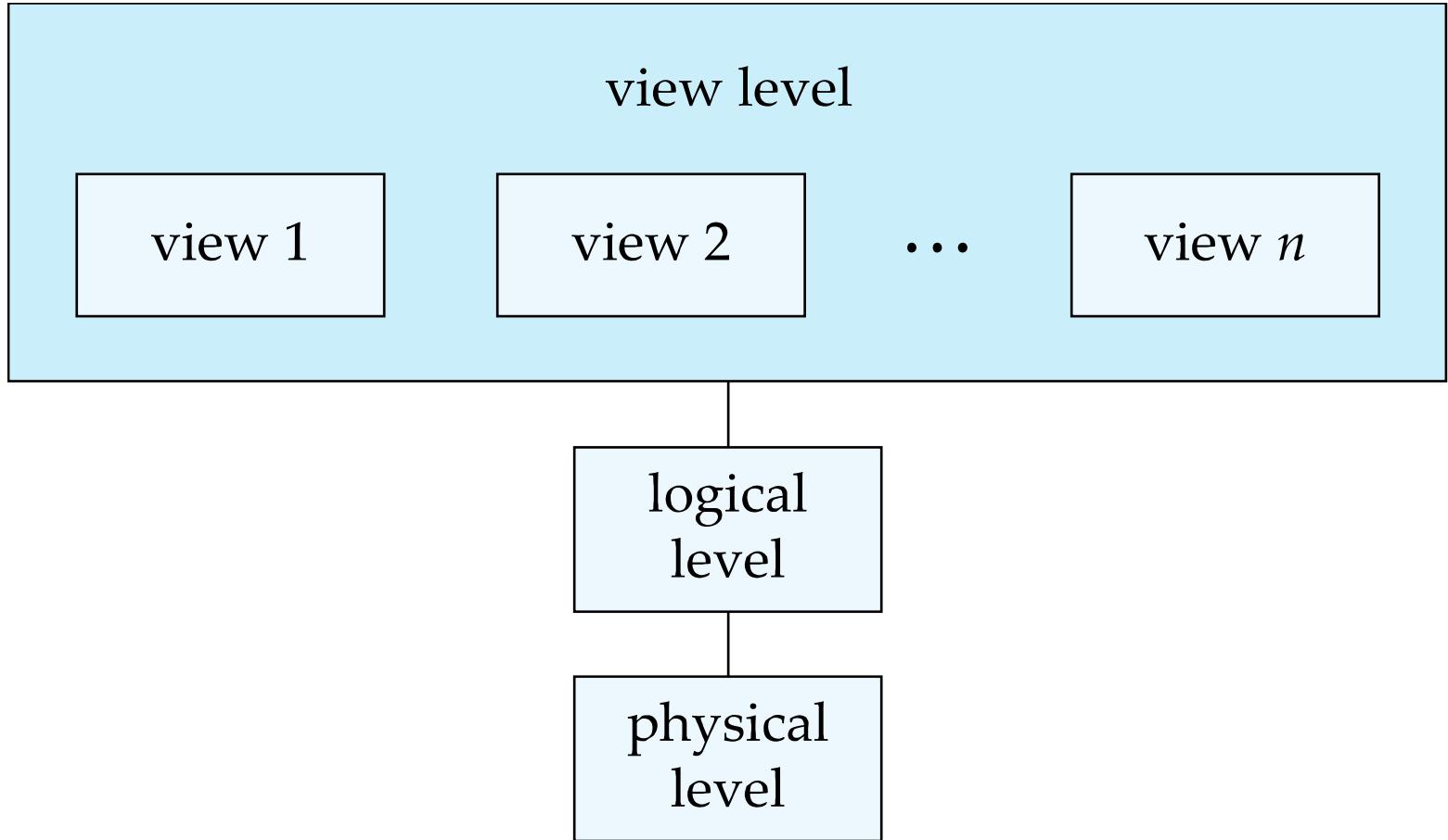
Levels of Abstraction

- **Physical level:** describes how a record (e.g., instructor) is stored.
- **Logical level:** describes data stored in database, and the relationships among the data.
- **View level:** application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.



View of Data

An architecture for a database system





Example

```
type instructor = record
    ID : string;
    name : string;
    dept_name : string;
    salary : integer;
end;
```

- A university organization may have several such record types:
 - Department (dept name, building, and budget)
 - Course (course id, title, dept name, and credits)
 - Student (ID , name, dept name, and tot_cred)



Data Models

- A collection of tools for describing
 - Data
 - Data relationships
 - Data semantics
 - Data constraints

- Data Models:
 - Relational model
 - Entity-Relationship data model (mainly for database design)
 - Complex Types
 - ▶ Object-based data models (Object-oriented and Object-relational)
 - ▶ Semi-structured data model



Relational Model

- All the data is stored in various tables.
- Example of tabular data in the relational model

The diagram shows a table with four columns labeled *ID*, *name*, *dept_name*, and *salary*. There are 12 rows of data. Two arrows point to the right from the top of the table: one pointing to the *dept_name* column and another pointing to the bottom-left of the table body, both labeled "Columns". A single arrow points to the left from the middle-right of the table body, labeled "Rows".

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table



Example of Relational Database

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table



Topics

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Structured Query Language (SQL)
- Database Design Approaches
 - Entity Relationship (ER) Model
 - Normalization
- Advanced Topics (Complex Data Types):
 - JSON
 - XML
 - RDF
 - Object-based data models



Question?



Database Systems

Lecture 2: Introduction

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book



Outline

- **Schemas and Instances**
- Data Definition Language and Data Manipulation Language
- Database Design
- Database Engine
- Database Architecture
- Data Mining and Information Retrieval
- Data Models
- History of Database Systems



Schemas and Instances

- Similar to types and variables in programming languages
- **Schema**
 - **Physical schema** – the overall physical structure of the database
 - **Logical Schema** – the overall logical structure of the database
 - Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
 - Analogous to type information of a variable in a program
- **Instance** – the actual content of the database at a particular point in time
 - Analogous to the value of a variable



Instances and Schemas

- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
 - Applications depend on the logical schema
 - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.



Outline

- Schemas and Instances
- **Data Definition Language and Data Manipulation Language**
- Database Design
- Database Engine
- Database Architecture
- Data Mining and Information Retrieval
- Data Models
- History of Database Systems



Data Definition Language (DDL)

- Specification notation for defining the database schema
- DDL compiler generates a set of table templates stored in a *data dictionary*
- Data dictionary contains metadata (i.e., data about data)
- A special type of table that can only be accessed and updated by the database system itself (not a regular user)



Data Definition Language (DDL)

- The data values stored in the database must satisfy certain consistency constraints
- Database systems implement these constraints that can be tested with minimal overhead
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Authorization



Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
 - DML also known as query language
- Different types of access:
 - Retrieval of information stored in the database
 - Insertion of new information into the database
 - Deletion of information from the database
 - Modification of information stored in the database



Data Manipulation Language (DML)

- DML types:
 - Procedural DMLs require a user to specify what data are needed and how to get those data
 - Declarative DMLs (nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data
- Two classes of languages
 - **Pure** – used for proving properties about computational power and for optimization
 - Relational Algebra
 - Tuple relational calculus
 - Domain relational calculus
 - **Commercial** – used in commercial systems
 - SQL is the most widely used commercial language



SQL

- The most widely used commercial language
- To be able to compute complex functions SQL is usually embedded in some higher-level language
- Application programs generally access databases through one of
 - Language extensions to allow embedded SQL
 - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database



DDL and DML Examples

■ DDL

```
create table instructor (
    ID      char(5),
    name    varchar(20),
    dept_name varchar(20),
    salary   numeric(8,2))
```

■ DML

```
select instructor.name
from instructor
where instrctor.dept_name = 'History';
```

```
select instructor. ID , department.dept_name
from instructor, department
where instructor.dept_name = department.dept_name
        and department.budget > 95000;
```



Outline

- Schemas and Instances
- Data Definition Language and Data Manipulation Language
- **Database Design**
- Database Engine
- Database Architecture
- Data Mining and Information Retrieval
- Data Models
- History of Database Systems



Database Design

The process of designing the general structure of the database:

- Logical Design – Deciding on the database schema.
Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database



Database Design

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table



Database Design

- Is there any problem with this relation?

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000



Design Approaches

- Need to come up with a methodology to ensure that each of the relations in the database is “good”
- Two ways of doing so:
 - Entity Relationship Model (Chapter 7)
 - Models an enterprise as a collection of *entities* and *relationships*
 - Represented diagrammatically by an *entity-relationship diagram*
 - Normalization Theory (Chapter 8)
 - Formalize what designs are bad, and test for them



Design Approaches

■ Specification of functional requirements

- Users describe the kinds of operations (or transactions) that will be performed on the data
 - Modifying or updating data
 - Searching for and retrieving specific data
 - Deleting data
- The designer can review the schema to ensure it meets functional requirements



Database Design for a University Organization

■ General description

- The university is organized into departments. Each department is identified by a unique name (dept_name), is located in a particular building, and has a budget.
- Each department has a list of courses it offers. Each course has associated with a course id, title, dept_name, and credits, and may also have associated prerequisites.
- Instructors are identified by their unique ID . Each instructor has name, associated department (dept_name), and salary.
- Students are identified by their unique ID . Each student has a name, an associated major department (dept_name), and tot_cred (total credit hours the student earned thus far)



Database Design for a University Organization

■ General description (cont.)

- The university maintains a list of classrooms, specifying the name of the building, room number, and room capacity.
- The university maintains a list of all classes (sections) taught. Each section is identified by a course_id, sec_id, year, and semester, and has associated with it a semester, year, building, room_number, and time_slot_id (the time slot when the class meets).
- The department has a list of teaching assignments specifying, for each instructor, the sections the instructor is teaching.
- The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).



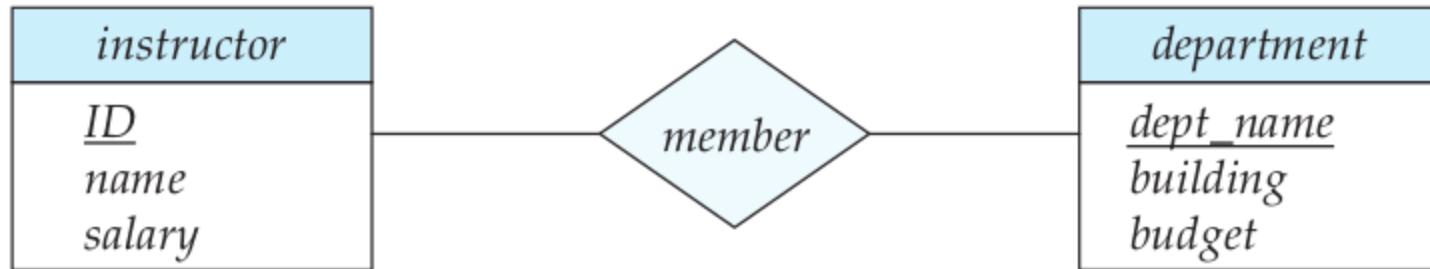
The Entity-Relationship Model

- Uses a collection of
 - Entities (basic objects)
 - Relationships among these objects
- Entities are described in a database by a set of attributes
- Example:
 - Entity: instructor
 - Attributes: ID , name, and salary
 - The extra attribute ID is used to identify an instructor uniquely (since it may be possible to have two instructors with the same name and the same salary).
 - Relationship: member (associates an instructor with his/her department)



The Entity-Relationship Model

- Entity-relationship (E-R) diagram expresses the overall logical structure (schema) of a database
 - Unified Modeling Language (UML)





Normalization

- Goal:
 - Designing schemas that are in an appropriate normal form.
- Generating a set of relation schemas that allows us
 - Storing information without unnecessary redundancy
 - Retrieving information easily



Database Design

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table



Database Design

- Is there any problem with this relation?

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000



Normalization

■ Problems of a bad design

- Repetition of information
- Inability to represent certain information



Outline

- Schemas and Instances
- Data Definition Language and Data Manipulation Language
- Database Design
- **Database Engine**
- Database Architecture
- Data Mining and Information Retrieval
- Data Models
- History of Database Systems



Database Engine

- Storage manager
- Query processing
- Transaction manager



Storage Management

- **Storage manager** is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

- The storage manager is responsible to the following tasks:
 - Interaction with the OS file manager
 - Efficient storing, retrieving and updating of data



Storage Management

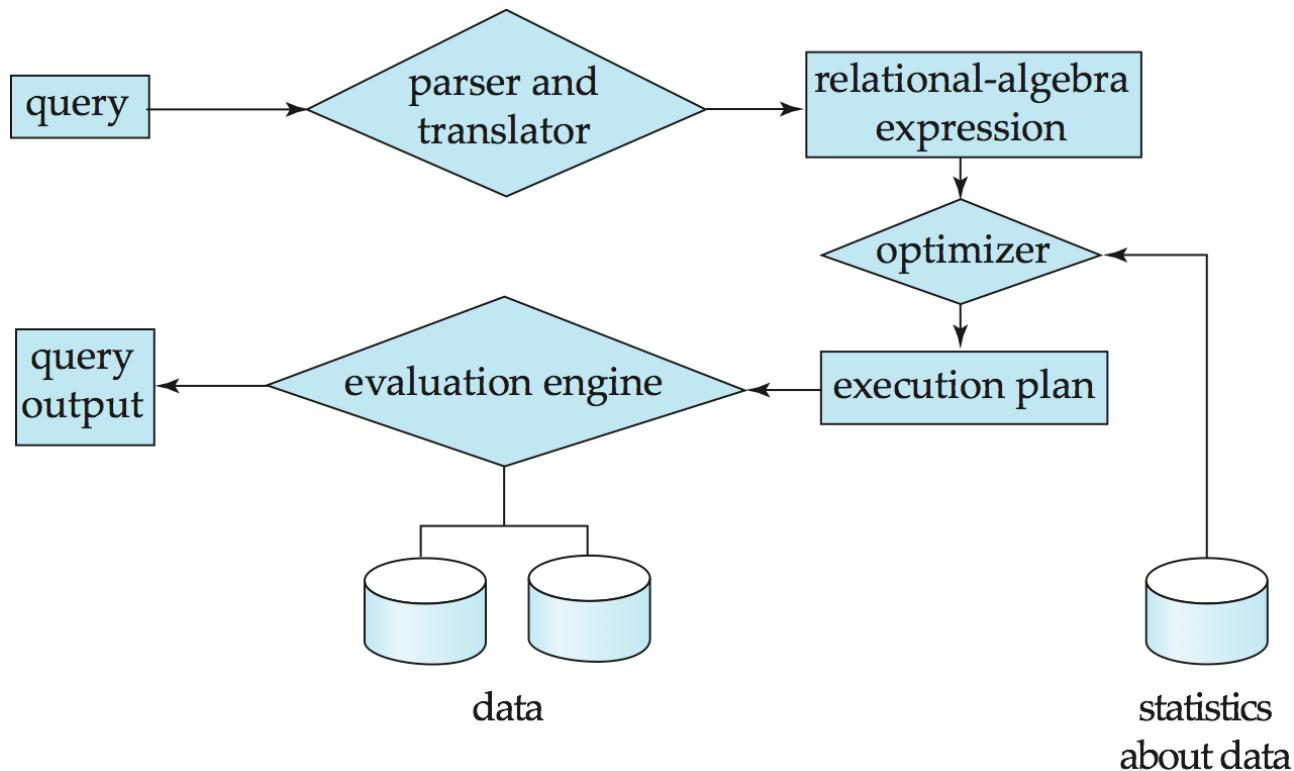
- The storage manager components include:
 - Authorization and integrity manager
 - Transaction manager
 - File manager
 - Buffer manager

- Data structures:
 - Data files
 - Data dictionary
 - Indices



Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Query Processing

- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
 - Depends critically on statistical information about relations which the database must maintain
 - Need to estimate statistics for intermediate results to compute cost of complex expressions



Transaction Management

- What if the system fails?
- What if more than one user is concurrently updating the same data?
- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
 - Recovery manager detects system failures and restore the database to the state that existed prior to the occurrence of the failure.
 - Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

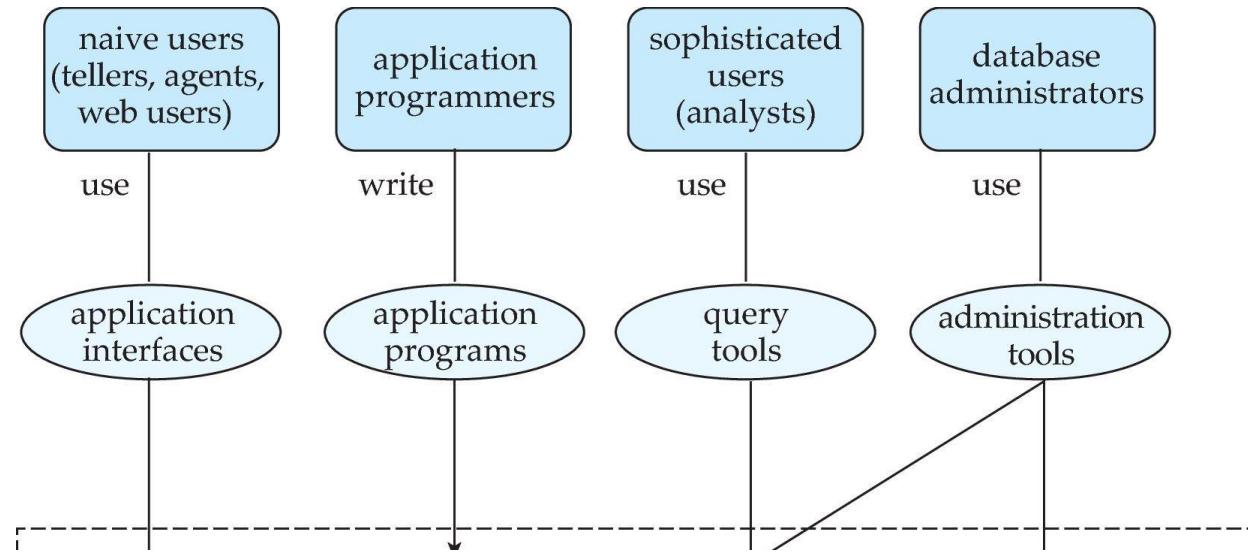


Outline

- Schemas and Instances
- Data Definition Language and Data Manipulation Language
- Database Design
- Database Engine
- **Database Architecture**
- Data Mining and Information Retrieval
- Data Models
- History of Database Systems



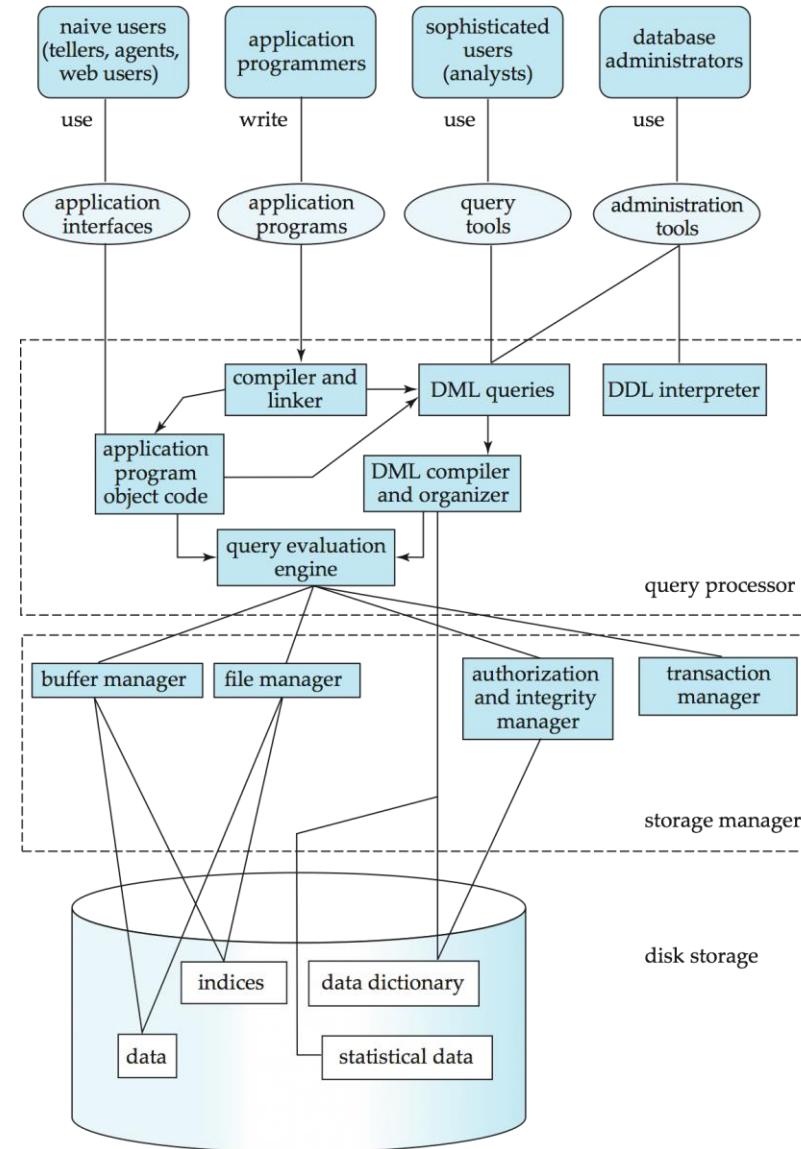
Database Users and Administrators



Database



Database System Internals





Database Architecture

- The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:
 - Centralized
 - Client-server
 - Parallel (multi-processor)
 - Distributed



Outline

- Schemas and Instances
- Data Definition Language and Data Manipulation Language
- Database Design
- Database Engine
- Database Architecture
- **Data Mining and Information Retrieval**
- Data Models
- History of Database Systems



Data Mining and Information Retrieval

- **Data mining**: the process of semi-automatically analyzing large databases to find useful patterns.
 - Also known as “knowledge discovery in databases”
-
- **Information retrieval**: querying of unstructured textual data
 - Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage



Outline

- Schemas and Instances
- Data Definition Language and Data Manipulation Language
- Database Design
- Database Engine
- Database Architecture
- Data Mining and Information Retrieval
- **Data Models**
- History of Database Systems



Data Models

■ Other Data Models

- Object-Based Data Models
- Semi-structured Data Models



Object-Relational Data Models

- Relational model: flat, “atomic” values
- Object Relational Data Models
 - Extend the relational data model by including object orientation and constructs to deal with added data types.
 - Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
 - Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
 - Provide upward compatibility with existing relational languages.



Semi-structured Data Models

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents
- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data



Outline

- Schemas and Instances
- Data Definition Language and Data Manipulation Language
- Database Design
- Database Engine
- Database Architecture
- Data Mining and Information Retrieval
- Data Models
- **History of Database Systems**



History of Database Systems

- 1950s and early 1960s:
 - Data processing using magnetic tapes for storage
 - Tapes provided only sequential access
 - Punched cards for input
- Late 1960s and 1970s:
 - Hard disks allowed direct access to data
 - Network and hierarchical data models in widespread use
 - Ted Codd defines the relational data model
 - High-performance (for the era) transaction processing
- 1980s:
 - Research relational prototypes evolve into commercial systems
 - SQL becomes industrial standard
 - Parallel and distributed database systems
 - Object-oriented database systems



History of Database Systems

- 1990s:
 - Large decision support and data-mining applications
 - Large multi-terabyte data warehouses
 - Emergence of Web commerce
- Early 2000s:
 - XML and XQuery standards
 - Automated database administration
- Later 2000s:
 - Giant data storage systems
 - Google BigTable, Yahoo PNuts, Amazon, ..



Questions?



Database Systems

Lecture 3: Intro to Relational Model

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book



- **Structure of Relational Databases**
- Database Schema
- Keys
- Schema Diagrams
- Relational Query Languages
- The Relational Algebra



Example of a Relation

Diagram illustrating the components of a relation:

- attributes (or columns):** Indicated by three arrows pointing to the column headers *ID*, *name*, *dept_name*, and *salary*.
- tuples (or rows):** Indicated by three arrows pointing to the data rows in the table.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000



Attribute Types

- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain. Indicated that the value is “unknown”
- The null value causes complications in the definition of many operations



- Structure of Relational Databases
- **Database Schema**
- Keys
- Schema Diagrams
- Relational Query Languages
- The Relational Algebra



Relation Schema and Instance

- A_1, A_2, \dots, A_n are *attributes*
- $R = (A_1, A_2, \dots, A_n)$ is a *relation schema*

Example:

instructor = (*ID*, *name*, *dept_name*, *salary*)

- Formally, given sets D_1, D_2, \dots, D_n a **relation** r is a subset of $D_1 \times D_2 \times \dots \times D_n$
Thus, a relation is a set of n -tuples (a_1, a_2, \dots, a_n) where each $a_i \in D_i$
- The current values (**relation instance**) of a relation are specified by a table
- An element t of r is a *tuple*, represented by a *row* in a table



Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example: *instructor* relation with unordered tuples

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



Database Schema

- Database schema -- is the logical structure of the database.
- Database instance -- is a snapshot of the data in the database at a given instant in time.
- Example:
 - schema: *instructor (ID, name, dept_name, salary)*
 - Instance:

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



- Structure of Relational Databases
- Database Schema
- **Keys**
- Schema Diagrams
- Relational Query Languages
- The Relational Algebra



Keys

- Let $K \subseteq R$
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
 - Example: $\{ID\}$ and $\{ID, name\}$ are both superkeys of *instructor*.
- Superkey K is a **candidate key** if K is minimal
Example: $\{ID\}$ is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.

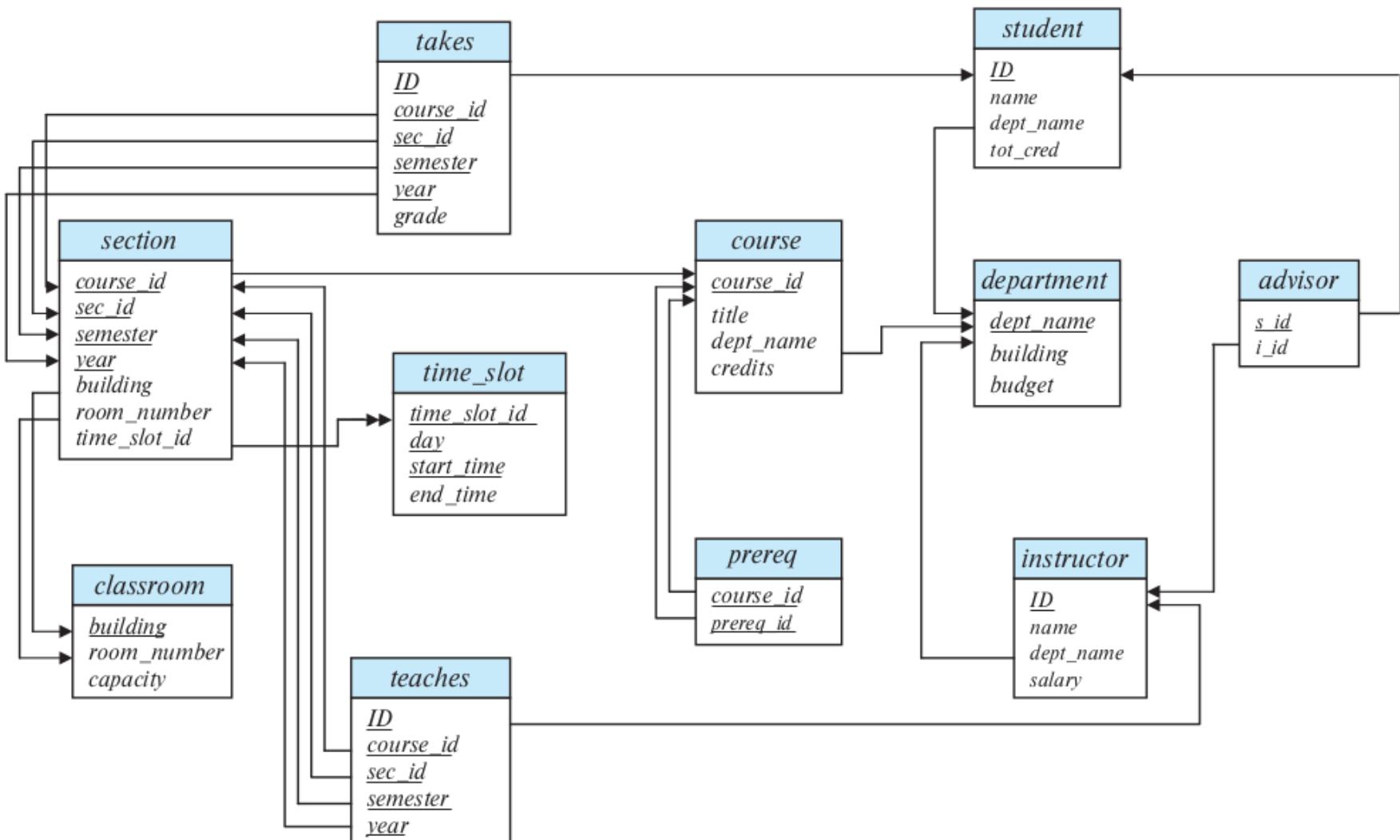
- **Foreign key** constraint: Value in one relation must appear in another
 - **Referencing** relation
 - **Referenced** relation
 - Example – *dept_name* in *instructor* is a foreign key from *instructor* referencing *department*



- Structure of Relational Databases
- Database Schema
- Keys
- **Schema Diagrams**
- Relational Query Languages
- The Relational Algebra



Schema Diagram for University Database





Example: instructor relation

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.1 The *instructor* relation.



Example: course relation

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.2 The *course* relation.



Example: prereq relation

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 2.3 The *prereq* relation.



Example: department relation

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.



Example: time_slot relation

<i>time_slot_id</i>	<i>day</i>	<i>start_time</i>	<i>end_time</i>
A	M	8:00	8:50
A	W	8:00	8:50
A	F	8:00	8:50
B	M	9:00	9:50
B	W	9:00	9:50
B	F	9:00	9:50
C	M	11:00	11:50
C	W	11:00	11:50
C	F	11:00	11:50
D	M	13:00	13:50
D	W	13:00	13:50
D	F	13:00	13:50
E	T	10:30	11:45
E	R	10:30	11:45
F	T	14:30	15:45
F	R	14:30	15:45
G	M	16:00	16:50
G	W	16:00	16:50
G	F	16:00	16:50
H	W	10:00	12:30

Figure A.12 The *time_slot* relation.



Example: section relation

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 2.6 The *section* relation.



Example: teaches relation

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 2.7 The *teaches* relation.



- Structure of Relational Databases
- Database Schema
- Keys
- Schema Diagrams
- **Relational Query Languages**
- The Relational Algebra



Relational Query Languages

- Procedural vs .non-procedural, or declarative
- “Pure” languages:
 - Relational algebra
 - Tuple relational calculus
 - Domain relational calculus
- The above 3 pure languages are equivalent in computing power
- We will concentrate in this chapter on relational algebra
 - Consists of 6 basic operations



- Structure of Relational Databases
- Database Schema
- Keys
- Schema Diagrams
- Relational Query Languages
- **The Relational Algebra**



Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
 - select: σ
 - project: Π
 - union: \cup
 - set difference: $-$
 - Cartesian product: \times
 - rename: ρ



Select Operation

- The **select** operation selects tuples that satisfy a given predicate.
- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Example: select those tuples of the *instructor* relation where the instructor is in the “Physics” department.
 - Query

$$\sigma_{dept_name = "Physics"}(instructor)$$

- Result

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
33456	Gold	Physics	87000



Select Operation (Cont.)

- We allow comparisons using
 $=, \neq, >, \geq, <, \leq$
in the selection predicate.
- We can combine several predicates into a larger predicate by using the connectives:
 \wedge (and), \vee (or), \neg (not)
- Example: Find the instructors in Physics with a salary greater \$90,000, we write:

$$\sigma_{dept_name = "Physics"} \wedge salary > 90,000 (instructor)$$

- The select predicate may include comparisons between two attributes.
 - Example, find all departments whose name is the same as their building name:

$$\sigma_{dept_name = building} (department)$$



Select Operation

selection of rows (tuples)

- Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D > 5}(r)$



Select Operation

selection of rows (tuples)

- Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10



Example: select

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

Figure 2.10 Result of query selecting *instructor* tuples with salary greater than \$85000.



Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:

$$\prod_{A_1, A_2, A_3, \dots, A_k} (r)$$

where A_1, A_2, \dots, A_k are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets



Project Operation Example

- Example: eliminate the *dept_name* attribute of *instructor*
- Query:

$$\prod_{ID, name, salary} (instructor)$$

- Result:

<i>ID</i>	<i>name</i>	<i>salary</i>
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000



Project Operation selection of columns (Attributes)

- Relation r :

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\Pi_{A,C}(r)$



Project Operation selection of columns (Attributes)

- Relation r :

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\Pi_{A,C}(r)$

$$\begin{array}{|c|c|} \hline A & C \\ \hline \alpha & 1 \\ \hline \alpha & 1 \\ \hline \beta & 1 \\ \hline \beta & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A & C \\ \hline \alpha & 1 \\ \hline \beta & 1 \\ \hline \beta & 2 \\ \hline \end{array}$$



Example: project

<i>ID</i>	<i>salary</i>
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

Result of query selecting attributes *ID* and *salary* from the *instructor* relation.



Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query -- Find the names of all instructors in the Physics department.

$$\prod_{name}(\sigma_{dept_name = "Physics"}(instructor))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.



Example: select and project

<i>ID</i>	<i>salary</i>
12121	90000
22222	95000
33456	87000
83821	92000

Result of selecting attributes *ID* and *salary* of instructors with salary greater than \$85,000.



Union Operation

- The union operation allows us to combine two relations
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
 1. r, s must have the *same arity* (same number of attributes)
 2. The attribute domains must be compatible (example: 2nd column of r deals with the same type of values as does the 2nd column of s)
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\begin{aligned} & \prod_{course_id} (\sigma_{semester="Fall"} \wedge year=2017)(section) \cup \\ & \prod_{course_id} (\sigma_{semester="Spring"} \wedge year=2018)(section) \end{aligned}$$



Union Operation (Cont.)

- Result of:

$$\begin{aligned} & \prod_{course_id} (\sigma_{semester="Fall"} \wedge year=2017)(section) \cup \\ & \prod_{course_id} (\sigma_{semester="Spring"} \wedge year=2018)(section) \end{aligned}$$

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101



Union of two relations

- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cup s$:



Union of two relations

- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cup s$:

A	B
α	1
α	2
β	1
β	3



Set Difference Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation $r - s$
- Set differences must be taken between **compatible** relations.
 - r and s must have the **same arity**
 - attribute domains of r and s must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\prod_{course_id} (\sigma_{semester="Fall" \wedge year=2017}(section)) - \\ \prod_{course_id} (\sigma_{semester="Spring" \wedge year=2018}(section))$$

course_id
CS-347
PHY-101



Set difference of two relations

- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r - s$:



Set difference of two relations

- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r - s$:

A	B
α	1
β	1



Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: $r \cap s$
- Assume:
 - r, s have the *same arity*
 - attributes of r and s are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\begin{array}{l} \prod_{course_id} (\sigma_{semester="Fall"} \wedge year=2017(section)) \cap \\ \prod_{course_id} (\sigma_{semester="Spring"} \wedge year=2018(section)) \end{array}$$

course_id
CS-101



Set intersection of two relations

- Relation r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$



Set intersection of two relations

- Relation r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$

A	B
α	2



Cartesian-Product Operation

- The Cartesian-product operation (denoted by \times) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations *instructor* and *teaches* is written as:

instructor \times *teaches*

- We construct a tuple of the result out of each possible pair of tuples: one from the *instructor* relation and one from the *teaches* relation (see next slide)
- Since the instructor *ID* appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
 - *instructor.ID*
 - *teaches.ID*



The *instructor X teaches table*

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...
...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...
...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...
...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...
...



Cartesian-product

- Relations r, s :

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- $r \times s$:



Cartesian-product

- Relations r, s :

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- $r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b



Cartesian-product – naming issue

- Relations r, s :

A	B	
α	1	
β	2	
		r
B	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

- $r \times s$:

A	$r.B$	$s.B$	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b



The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ , is provided for that purpose
- The expression:

$$\rho_x(E)$$

returns the result of expression E under the name x

- Another form of the rename operation:

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$



Renaming a Table

- Allows us to refer to a relation, (say E) by more than one name.

$$\rho_x(E)$$

returns the expression E under the name X

- Relations r

A	B
α	1
β	2

r

- $r \times \rho_s(r)$

$r.A$	$r.B$	$s.A$	$s.B$
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2



Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C} (r \times s)$

- $r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- $\sigma_{A=C} (r \times s)$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b



Join Operation

- The Cartesian-Product

instructor X teaches

associates every tuple of instructor with every tuple of teaches.

- Most of the resulting rows have information about instructors who did NOT teach a particular course.

- To get only those tuples of “*instructor X teaches*” that pertain to instructors and the courses that they taught, we write:

$\sigma_{instructor.id = teaches.id} (instructor \times teaches)$

- We get only those tuples of “*instructor X teaches*” that pertain to instructors and the courses that they taught.
- The result of this expression, shown in the next slide



Join Operation (Cont.)

- The table corresponding to:

$$\sigma_{instructor.id = teaches.id} (instructor \times teaches)$$

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017



Joining two relations – Natural Join

- Let r and s be relations on schemas R and S respectively.

Then, the “natural join” of relations R and S is a relation on schema $R \cup S$ obtained as follows:

- Consider each pair of tuples t_r from r and t_s from s .
- If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
 - t has the same value as t_r on r
 - t has the same value as t_s on s



Natural Join Example

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- Natural Join

- $r \bowtie s$

$$\Pi_{A, r.B, C, r.D, E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$



Natural Join Example

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- Natural Join
 - $r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

$$\prod_{A, r.B, C, r.D, E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$



Example: natural join

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Result of natural join of the *instructor* and *department* relations.



The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by \leftarrow and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

$$Physics \leftarrow \sigma_{dept_name = "Physics"}(instructor)$$
$$Music \leftarrow \sigma_{dept_name = "Music"}(instructor)$$
$$Physics \cup Music$$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.



Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
- Query 1

$$\sigma_{dept_name = "Physics"} \wedge salary > 90,000 (instructor)$$

- Query 2
- $$\sigma_{dept_name = "Physics"} (\sigma_{salary > 90.000} (instructor))$$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.



Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department
- Query 1

$$\sigma_{dept_name = "Physics"}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

- Query 2
- $$(\sigma_{dept_name = "Physics"}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.



Notes about Relational Languages

- Each Query input is a table (or set of tables)
- Each query output is a table.
- All data in the output table appears in one of the input tables
- Can we compute:
 - SUM
 - AVG
 - MAX
 - MIN



Summary of Relational Algebra Operators

Symbol (Name)	Example of Use
σ (Selection)	$\sigma \text{ salary} >= 85000 \text{ (instructor)}$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi ID, \text{salary} \text{ (instructor)}$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
\times (Cartesian Product)	$\text{instructor} \times \text{department}$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
\cup (Union)	$\Pi name \text{ (instructor)} \cup \Pi name \text{ (student)}$ Output the union of tuples from the two input relations.
$-$ (Set Difference)	$\Pi name \text{ (instructor)} -- \Pi name \text{ (student)}$ Output the set difference of tuples from the two input relations.
\bowtie (Natural Join)	$\text{instructor} \bowtie \text{department}$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.



Questions?



Database Systems

Lecture 4: Introduction to SQL

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book



Outline

- **Overview of The SQL Query Language**
- **Data Definition**
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database



History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999
 - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
 - Not all examples here may work on your particular system.



Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And as we will see later, also other information such as
 - The set of indices to be maintained for each relations.
 - Security and authorization information for each relation.
 - The physical storage structure of each relation on disk.



Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length n .
- **varchar(n).** Variable length character strings, with user-specified maximum length n .
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., **numeric(3,1)**, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least n digits.
- More are covered in Chapter 4.



Create Table Construct

- An SQL relation is defined using the **create table** command:

```
create table r (A1 D1, A2 D2, ..., An Dn,  
                  (integrity-constraint1),  
                  ...,  
                  (integrity-constraintk))
```

- r is the name of the relation
- each A_i is an attribute name in the schema of relation r
- D_i is the data type of values in the domain of attribute A_i

- Example:

```
create table instructor (  
    ID          char(5),  
    name        varchar(20),  
    dept_name   varchar(20),  
    salary      numeric(8,2))
```



Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, \dots, A_n)
- **foreign key** (A_m, \dots, A_n) **references** r

Example:

```
create table instructor (
    ID          char(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    salary      numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department);
```

primary key declaration on an attribute automatically ensures **not null**



And a Few More Relation Definitions

```
create table department(  
    dep_name      varchar(20),  
    building      varchar(15),  
    budget        numeric(12,2),  
    primary key (dept_name));
```



And a Few More Relation Definitions

```
create table course (
    course_id      varchar(8),
    title          varchar(50),
    dept_name      varchar(20),
    credits         numeric(2,0),
    primary key (course_id),
    foreign key (dept_name) references department);
```



And a Few More Relation Definitions

```
create table section (
    course_id      varchar(8),
    sec_id         varchar(8),
    semester       varchar(6),
    year           numeric(4,0),
    building        varchar(15),
    room_number     varchar(7),
    time_slot_id   varchar(4),
    primary key (course_id, sec_id, semester, year),
    foreign key (course_id) references course);
```



And a Few More Relation Definitions

```
create table student (
    ID          varchar(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    tot_cred    numeric(3,0),
    primary key (ID),
    foreign key (dept_name) references department);
```



And a Few More Relation Definitions

```
create table takes (
    ID          varchar(5),
    course_id   varchar(8),
    sec_id      varchar(8),
    semester    varchar(6),
    year        numeric(4,0),
    grade       varchar(2),
    primary key (ID, course_id, sec_id, semester, year) ,
    foreign key (ID) references student,
    foreign key (course_id, sec_id, semester, year)
        references section);
```



And a Few More Relation Definitions

```
create table takes (
    ID          varchar(5),
    course_id   varchar(8),
    sec_id      varchar(8),
    semester    varchar(6),
    year        numeric(4,0),
    grade       varchar(2),
    primary key (ID, course_id, sec_id, semester, year) ,
    foreign key (ID) references student,
    foreign key (course_id, sec_id, semester, year)
        references section);
```

- Note: `sec_id` can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester



Updates to tables

■ Insert

- **insert into *instructor* values ('10211', 'Smith', 'Biology', 66000);**

■ Delete

- Remove all tuples from the *student* relation
 - **delete from *student***

■ Drop Table

- **drop table *r***

■ Alter

- **alter table *r* add *A D***
 - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.
 - All existing tuples in the relation are assigned *null* as the value for the new attribute.
- **alter table *r* drop *A***
 - where *A* is the name of an attribute of relation *r*
 - Dropping of attributes not supported by many databases.



Outline

- Overview of The SQL Query Language
- Data Definition
- **Basic Query Structure**
- **Additional Basic Operations**
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database



Basic Query Structure

- A typical SQL query has the form:

```
select A1, A2, ..., An  
from r1, r2, ..., rm  
where P
```

- A_i represents an attribute
 - R_j represents a relation
 - P is a predicate.
- The result of an SQL query is a relation.



The select Clause

- The **select** clause lists the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra

- Example: find the names of all instructors:

```
select name  
from instructor
```

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
 - E.g., *Name* \equiv *NAME* \equiv *name*
 - Some people use upper case wherever we use bold font.



Example: select clause

<i>name</i>
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of “select *name* from *instructor*”.



The select Clause

- SQL allows duplicates in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the department names of all instructors, and remove duplicates

```
select distinct dept_name
from instructor
```

- The keyword **all** specifies that duplicates should not be removed.

```
select all dept_name
from instructor
```



Example: select clause

<i>dept_name</i>
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of “select *dept_name* from *instructor*”.



The select Clause

- An asterisk in the select clause denotes “all attributes”

```
select *  
from instructor
```



The select Clause

- An attribute can be a literal with no **from** clause

```
select '437'
```

- Results is a table with one column and a single row with value “437”
- Can give the column a name using:

```
select '437' as FOO
```



The select Clause

- An attribute can be a literal with **from** clause

```
select 'A'  
from instructor
```

- Result is a table with one column and N rows (number of tuples in the *instructors* table), each row with value “A”



The select Clause

- The **select** clause can contain arithmetic expressions involving the operation, +, −, *, and /, and operating on constants or attributes of tuples.

- The query:

```
select ID, name, salary/12
      from instructor
```

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename “*salary/12*” using the **as** clause:

```
select ID, name, salary/12 as monthly_salary
```



The where Clause

- The **where** clause specifies conditions that the result must satisfy
 - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

```
select name  
from instructor  
where dept_name = 'Comp. Sci.'
```



The where Clause

- Comparison results can be combined using the logical connectives **and**, **or**, and **not**
 - To find all instructors in Comp. Sci. dept with salary > 70000



The where Clause

- Comparison results can be combined using the logical connectives **and**, **or**, and **not**
 - To find all instructors in Comp. Sci. dept with salary > 70000

```
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 70000
```

- Comparisons can be applied to results of arithmetic expressions.



Example: where clause

<i>name</i>
Katz
Brandt

Figure 3.4 Result of “Find the names of all instructors in the Computer Science department who have salary greater than \$70,000.”



The from Clause

- The **from** clause lists the relations involved in the query
 - Corresponds to the Cartesian product operation of the relational algebra.

- Find the Cartesian product *instructor X teaches*

```
select *  
from instructor, teaches
```

- generates every possible instructor – teaches pair, with all attributes from both relations.
 - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
-
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).



Cartesian Product

instructor

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
...

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Inst.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...
...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Pinance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Pinance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Pinance	90000	22222	PHY-101	1	Fall	2009
...
...



Example: from clause

- Find the names of all instructors who have taught some course and the course_id

```
select *
from instructor , teaches
where instructor.ID = teaches.ID
```



Example: from Clause

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.8 The natural join of the *instructor* relation with the *teaches* relation.



Example: from clause

- Find all instructors who have taught some course

```
select *
from instructor , teaches
where instructor.ID = teaches.ID
```

- This query can be written more concisely using the natural-join operation in SQL as:

```
select *
from instructor natural join teaches
```



Example: from clause

- Find the names of all instructors who have taught some course and the course_id

```
select name, course_id  
from instructor , teaches  
where instructor.ID = teaches.ID
```

```
select name, course_id  
from instructor natural join teaches
```



Example: from Clause

<i>name</i>	<i>course_id</i>
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.7 Result of “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.”



Examples: from clause

- Find the names of all instructors who have taught some course and the course_id

```
select name, course_id  
from instructor , teaches  
where instructor.ID = teaches.ID
```

- Find the names of all instructors in the Art department who have taught some course and the course_id

```
select name, course_id  
from instructor , teaches  
where instructor.ID = teaches.ID  
and instructor.dept_name = 'Art'
```



The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

old-name as new-name

- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.

```
select distinct T.name  
from instructor as T, instructor as S  
where T.salary > S.salary and S.dept_name = 'Comp. Sci.'
```

- Keyword **as** is optional and may be omitted

instructor as T ≡ instructor T



Cartesian Product Example

- Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of “Bob”



Cartesian Product Example

- Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of “Bob”

```
Select supervisor  
from emp-super  
where person= 'Bob'
```



Cartesian Product Example

- Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of the supervisor of “Bob”



Cartesian Product Example

- Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of the supervisor of “Bob”

```
Select B.supervisor  
from emp-super as A, emp-super as B  
where A.supervisor = B.person and A.person = 'Bob'
```



String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring “dar”.

```
select name  
from instructor  
where name like '%dar%'
```

- Match the string “100%”

```
like '100 \%' escape '\'
```

in that above we use backslash (\) as the escape character.



String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
 - ‘Intro%’ matches any string beginning with “Intro”.
 - ‘%Comp%’ matches any string containing “Comp” as a substring.
 - ‘___’ matches any string of exactly three characters.
 - ‘___ %’ matches any string of at least three characters.
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.



Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

```
select distinct name  
from   instructor  
order by name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

```
order by name desc
```

- Can sort on multiple attributes

```
order by dept_name, name
```



Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq \$90,000$ and $\leq \$100,000$)

```
select name  
from instructor  
where salary between 90000 and 100000
```

- Tuple comparison

```
select name, course_id  
from instructor, teaches  
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```



Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2 :
 1. $\sigma_\theta(r_1)$: If there are c_1 copies of tuple t_1 in r_1 , and t_1 satisfies selections σ_θ , then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
 2. $\Pi_A(r)$: For each copy of tuple t_1 in r_1 , there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1 .
 3. $r_1 \times r_2$: If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2 , there are $c_1 \times c_2$ copies of the tuple $t_1 \cdot t_2$ in $r_1 \times r_2$



Duplicates (Cont.)

- Example: Suppose multiset relations $r_1 (A, B)$ and $r_2 (C)$ are as follows:

$$r_1 = \{(1, a) (2, a)\} \quad r_2 = \{(2), (3), (3)\}$$

- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, while $\Pi_B(r_1) \times r_2$ would be $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$
- SQL duplicate semantics:

```
select A1, A2, ..., An  
from r1, r2, ..., rm  
where P
```

is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$



Outline

- Overview of The SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- **Set Operations**
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database



Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

```
(select course_id  
from section  
where sem = 'Fall' and year = 2009)  
union  
(select course_id  
from section  
where sem = 'Spring' and year = 2010)
```



Set Operations

- Find courses that ran in Fall 2009 and in Spring 2010

```
(select course_id  
from section  
where sem = 'Fall' and year = 2009)  
intersect  
(select course_id  
from section  
where sem = 'Spring' and year = 2010)
```



Set Operations

- Find courses that ran in Fall 2009 but not in Spring 2010

```
(select course_id  
from section  
where sem = 'Fall' and year = 2009)  
except  
(select course_id  
from section  
where sem = 'Spring' and year = 2010)
```



Example: set operations

<i>course_id</i>
CS-101
CS-347
PHY-101

Figure 3.9 The *c1* relation, listing courses taught in Fall 2009.

<i>course_id</i>
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Figure 3.10 The *c2* relation, listing courses taught in Spring 2010.



Example: set operations

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 3.11 The result relation for $c1 \cup c2$.

course_id
CS-101

Figure 3.12 The result relation for $c1 \cap c2$.

course_id
CS-347
PHY-101

Figure 3.13 The result relation for $c1 - c2$.



Set Operations

- Find the salaries of all instructors that are less than the largest salary.

```
select distinct T.salary
from instructor as T, instructor as S
where T.salary < S.salary
```

- Find all the salaries of all instructors

```
select distinct salary
from instructor
```

- Find the largest salary of all instructors.

```
(select “second query” )
except
(select “first query”)
```



Set Operations

- Set operations **union**, **intersect**, and **except**
 - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.
- Suppose a tuple occurs m times in r and n times in s , then, it occurs:
 - $m + n$ times in $r \text{ union all } s$
 - $\min(m,n)$ times in $r \text{ intersect all } s$
 - $\max(0, m - n)$ times in $r \text{ except all } s$



Outline

- Overview of The SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- **Null Values**
- Aggregate Functions
- Nested Subqueries
- Modification of the Database



Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
 - Example: $5 + \text{null}$ returns null
- The predicate **is null** can be used to check for null values.
 - Example: Find all instructors whose salary is null.

```
select name  
from instructor  
where salary is null
```



Null Values and Three Valued Logic

- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown*
 - Example: $5 < \text{null}$ or $\text{null} < > \text{null}$ or $\text{null} = \text{null}$
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*



Null Values and Three Valued Logic

- Three-valued logic using the value *unknown*:
 - OR: $(\text{unknown} \text{ or } \text{true}) = \text{true}$,
 $(\text{unknown} \text{ or } \text{false}) = \text{unknown}$
 $(\text{unknown} \text{ or } \text{unknown}) = \text{unknown}$
 - AND: $(\text{true} \text{ and } \text{unknown}) = \text{unknown}$,
 $(\text{false} \text{ and } \text{unknown}) = \text{false}$,
 $(\text{unknown} \text{ and } \text{unknown}) = \text{unknown}$
 - NOT: $(\text{not unknown}) = \text{unknown}$
- “ P is **unknown**” evaluates to true if predicate P evaluates to *unknown*



Outline

- Overview of The SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- **Aggregate Functions**
- Nested Subqueries
- Modification of the Database



Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values



Aggregate Functions

- Find the average salary of instructors in the Computer Science department

```
select avg (salary)
from instructor
where dept_name= 'Comp. Sci.';
```



Aggregate Functions

- Find the total number of instructors who teach a course in the Spring 2010 semester

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```



Aggregate Functions

- Find the number of tuples in the *course* relation

```
select count (*)  
from course;
```



Aggregate Functions – Group By

- Find the average salary of instructors in each department

```
select dept_name, avg (salary) as avg_salary  
from instructor  
group by dept_name;
```

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000



Aggregation

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

```
/* erroneous query */  
select dept_name, ID, avg (salary)  
from instructor  
group by dept_name;
```



Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary)
from instructor
group by dept_name
having avg (salary) > 42000;
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups



Example: having clause

<i>dept_name</i>	<i>avg(avg_salary)</i>
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.17 The result relation for the query “Find the average salary of instructors in those departments where the average salary is more than \$42,000.”



Null Values and Aggregates

- Total all salaries

```
select sum (salary )  
from instructor
```

- Above statement ignores null amounts
- Result is *null* if there is no non-null amount

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes

- What if collection has only null values?
 - count returns 0
 - all other aggregates return null



Outline

- Overview of The SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- **Nested Subqueries**
- Modification of the Database



Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

```
select  $A_1, A_2, \dots, A_n$   
from  $r_1, r_2, \dots, r_m$   
where  $P$ 
```

as follows:

- A_i can be replaced by a subquery that generates a single value.
- r_i can be replaced by any valid subquery
- P can be replaced with an expression of the form:

$B <\text{operation}> (\text{subquery})$

Where B is an attribute and $<\text{operation}>$ to be defined later.



Subqueries in the Where Clause



Subqueries in the Where Clause

- A common use of subqueries is to perform tests:
 - For set membership
 - For set comparisons
 - For set cardinality.



Set Membership

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id in (select course_id
from section
where semester = 'Spring' and year= 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id not in (select course_id
from section
where semester = 'Spring' and year= 2010);
```



Set Membership

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
    (select course_id, sec_id, semester, year
     from teaches
     where teaches.ID= 10101);
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.



Set Comparison – “some” Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
              from instructor
              where dept name = 'Biology');
```



Definition of “some” Clause

- $F \text{ <comp> some } r \Leftrightarrow \exists t \in r \text{ such that } (F \text{ <comp> } t)$

Where <comp> can be: $<$, \leq , $>$, $=$, \neq

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 = \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$ (since $0 \neq 5$)

$(= \text{some}) \equiv \text{in}$

However, $(\neq \text{some}) \not\equiv \text{not in}$



Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name  
from instructor  
where salary > all (select salary  
                 from instructor  
                 where dept name = 'Biology');
```



Definition of “all” Clause

- $F \text{ <comp> all } r \Leftrightarrow \forall t \in r (F \text{ <comp> } t)$

$(5 < \text{all} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$

$(5 < \text{all} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$

$(5 = \text{all} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 \neq \text{all} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true} \text{ (since } 5 \neq 4 \text{ and } 5 \neq 6\text{)}$

$(\neq \text{all}) \equiv \text{not in}$

However, $(= \text{all}) \neq \text{in}$



Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$



Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```
select course_id
  from section as S
 where semester = 'Fall' and year = 2009 and
       exists (select *
                  from section as T
                 where semester = 'Spring' and year= 2010
                   and S.course_id = T.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query



Use of “not exists” Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name  
from student as S  
where not exists ( (select course_id  
                    from course  
                    where dept_name = 'Biology')  
                  except  
                  (select T.course_id  
                    from takes as T  
                    where S.ID = T.ID));
```

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants



Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2009

```
select T.course_id  
from course as T  
where unique (select R.course_id  
              from section as R  
              where T.course_id= R.course_id  
                and R.year = 2009);
```



Subqueries in the From Clause



Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than \$42,000."

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name) as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```



With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as  
    (select max(budget)  
     from department)  
select department.dep_name  
      from department, max_budget  
     where department.budget = max_budget.value;
```



Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total(dept_name, value) as
  (select dept_name, sum(salary)
   from instructor
   group by dept_name),
dept_total_avg(value) as
  (select avg(value)
   from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```



Subqueries in the Select Clause



Select clause

- Subqueries in select clause must return a scalar (single) value for each row returned by the outer query.
- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,  
       (select count(*)  
        from instructor  
       where department.dept_name = instructor.dept_name)  
      as num_instructors  
  from department,
```

- Runtime error if subquery returns more than one result tuple



Outline

- Overview of The SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- **Modification of the Database**



Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation



Deletion

- Delete all instructors

delete from *instructor*

- Delete all instructors from the Finance department

delete from *instructor*
where *dept_name*= 'Finance';

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

delete from *instructor*
where *dept name* **in** (**select** *dept name*
from *department*
where *building* = 'Watson');



Deletion

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor  
where salary < (select avg (salary)  
from instructor);
```

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
 1. First, compute **avg** (salary) and find all tuples to delete
 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)



Insertion

- Add a new tuple to *course*

insert into *course*

values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

insert into *course* (*course_id*, *title*, *dept_name*, *credits*)

values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

insert into *student*

values ('3003', 'Green', 'Finance', *null*);



Insertion

- Add all instructors to the *student* relation with tot_creds set to 0

```
insert into student
    select ID, name, dept_name, 0
        from instructor
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem



Updates

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others by a 5%
 - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

- The order is important
- Can be done better using the **case** statement (next slide)



Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor  
set salary = case  
    when salary <= 100000 then salary * 1.05  
    else salary * 1.03  
end
```



Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
  set tot_cred = (select sum(credits)
                   from takes, course
                  where takes.course_id = course.course_id and
                        S.ID= takes.ID.and
                        takes.grade <> 'F' and
                        takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum(credits)**, use:

```
case
  when sum(credits) is not null then sum(credits)
  else 0
end
```



Questions?



Database Systems

Lecture 5: Intermediate SQL

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book



Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization



Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause



Join operations – Example

■ Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

■ Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101



Joined Relations – Examples

- course **natural join** *prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101



Join operations – Example

■ Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

■ Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

■ Observe that

prereq information is missing for CS-315 and
course information is missing for CS-437



Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.



Left Outer Join

- course **natural left outer join** *prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>



Right Outer Join

- course **natural right outer join** *prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	<i>null</i>	<i>null</i>	<i>null</i>	CS-101



Full Outer Join

- course **natural full outer join** *prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>
CS-347	<i>null</i>	<i>null</i>	<i>null</i>	CS-101



Inner Join

- course **natural inner join** prereq

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prere_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

- The default join type, when the join clause is used without the outer prefix is the inner join.



Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<i>Join types</i>
inner join
left outer join
right outer join
full outer join

<i>Join Conditions</i>
natural
on <predicate>
using (A_1, A_1, \dots, A_n)



Joined Relations – Examples

- course **inner join** prereq **on**
 $course.course_id = prereq.course_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join?

- Alternative:

course, prereq **where**

$course.course_id = prereq.course_id$



Joined Relations – Examples

- course **left outer join** prereq **on**
course.course_id = prereq.course_id

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>	<i>course_id</i>
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	<i>null</i>	<i>null</i>



Joined Relations – Examples

- course **full outer join** prereq **using** (course_id)

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101



Outline

- Join Expressions
- **Views**
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization



Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

```
select ID, name, dept_name  
from instructor
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.



View Definition

- A view is defined using the **create view** statement which has the form

create view *v* as < query expression >

where <query expression> is any legal SQL expression. The view name is represented by *v*.



View Definition

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
 - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.



Example Views

- A view of instructors without their salary

```
create view faculty as
select ID, name, dept_name
from instructor
```

- Using views in SQL queries:
- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```



Example Views

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum (salary)
    from instructor
  group by dept_name;
```



Views Defined Using Other Views

- **create view physics_fall_2009 as**
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
and course.dept_name = 'Physics'
and section.semester = 'Fall'
and section.year = '2009';

- **create view physics_fall_2009_watson as**
select course_id, room_number
from physics_fall_2009
where building= 'Watson';



View Expansion

- Expand use of a view in a query/another view

```
create view physics_fall_2009_watson as
  (select course_id, room_number
   from (select course.course_id, building, room_number
          from course, section
          where course.course_id = section.course_id
          and course.dept_name = 'Physics'
          and section.semester = 'Fall'
          and section.year = '2009')
   where building= 'Watson';
```



Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.



View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat

 Find any view relation v_i in e_1

 Replace the view relation v_i by the expression defining v_i

until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate



Update of a View

- Views present serious problems if we express updates, insertions, or deletions with them.
- The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.
- Add a new tuple to *faculty* view which we defined earlier
insert into faculty values ('30765', 'Green', 'Music');

This insertion must be represented by the insertion of the tuple ('30765', 'Green', 'Music', null) into the *instructor* relation



Some Updates cannot be Translated Uniquely

- **create view *instructor_info* as**
select *ID*, *name*, *building*
from *instructor*, *department*
where *instructor.dept_name*= *department.dept_name*;

- **insert into *instructor_info* values ('69987', 'White', 'Taylor');**
 - which department, if multiple departments in Taylor?
 - what if no department is in Taylor?
 - what happen if we add the following tuples to the *instructor* and *department* relations?
('69987', 'White', null, null) into *instructor*
(null, 'Taylor', null) into *department*



Some Updates cannot be Translated Uniquely

- Most SQL implementations allow updates only on simple views
 - The **from** clause has only one database relation.
 - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
 - Any attribute not listed in the **select** clause can be set to null
 - The query does not have a **group by** or **having** clause.



And Some Not at All

- **create view** *history_instructors* **as**
select *
from *instructor*
where *dept_name*= 'History';

- What happens if we insert ('25566', 'Brown', 'Biology', 100000) into *history_instructors*?

NOTE:

By default, SQL would allow the above update to proceed. However, views can be defined with a check option clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's where clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the where clause conditions



Materialized Views

- **Materializing a view:** create a physical table containing all the tuples in the result of the query defining the view
 - Such views called **Materialized view**
- If relations used in the query are updated, the materialized view result becomes out of date
 - Need to **Maintain** the view, by updating the view whenever the underlying relations are updated.
 - The process of keeping the materialized view up-to-date is called **materialized view maintenance** (or just **view maintenance**)



Materialized Views Maintenance

- Maintaining a view can be done in different ways
 - View maintenance can be done immediately when any of the relations on which the view is defined is updated.
 - View maintenance can be performed lazily, when the view is accessed.
 - Some systems update materialized views only periodically



Outline

- Join Expressions
- Views
- **Transactions**
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization



Transactions

- Consists of a sequence of query and/or update statements.
- Atomic transaction
- Either fully executed or rolled back as if it never occurred

- Transactions begin implicitly and ended by one of the following
 - **Commit work** commits the current transaction
 - Making the updates performed by the transaction become permanent in the database.
 - After the transaction is committed, a new transaction is automatically started.
 - **Rollback work** causes the current transaction to be rolled back
 - It undoes all the updates performed by the SQL statements in the transaction.
 - Thus, the database state is restored to what it was before the first statement of the transaction was executed.



Transactions

- By default most databases commit each SQL statement automatically as a transaction
 - Can turn off auto commit for a session (e.g. using API)
 - In SQL:1999
 - **begin atomic end**
 - But not supported on most databases
- Further reading for transactions: Chapter 14



Outline

- Join Expressions
- Views
- Transactions
- **Integrity Constraints**
- SQL Data Types and Schemas
- Authorization



Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
 - A checking account must have a balance greater than \$10,000.00
 - A salary of a bank employee must be at least \$4.00 an hour
 - A customer must have a (non-null) phone number



Integrity Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check (P)**, where P is a predicate



Not Null and Unique Constraints

■ **not null**

- Declare *name* and *budget* to be **not null**

name varchar(20) not null

budget numeric(12,2) not null

■ **unique** (A_1, A_2, \dots, A_m)

- The unique specification states that the attributes A_1, A_2, \dots, A_m form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).



The check clause

- **check (P)**

where P is a predicate

Example: ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time_slot_id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```



Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



Cascading Actions in Referential Integrity

- **create table course (**
 course_id char(5) primary key,
 title varchar(20),
 dept_name varchar(20) references department)

- **create table course (**
 \dots
 dept_name varchar(20),
 foreign key (dept_name) references department
 on delete cascade
 on update cascade,
 \dots **)**

- alternative actions to cascade: **set null, set default**



Integrity Constraint Violation During Transactions

- E.g.

```
create table person (
    ID char(10),
    name char(40),
    mother char(10),
    father char(10),
    primary key ID,
    foreign key father references person,
    foreign key mother references person)
```

- How to insert a tuple without causing constraint violation ?
 - insert father and mother of a person before inserting person
 - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
 - OR defer constraint checking



Complex Check Clauses

- **check** (*time_slot_id* in (select *time_slot_id* from *time_slot*))
 - should be checked by any changes in *time_slot* table as well
- Every section has at least one instructor teaching the section.
 - how to write this?
- **create assertion** <assertion-name> **check** <predicate>;
 - introduce complex overhead
- Unfortunately: subquery in check clause not supported by pretty much any database
 - Alternative: triggers (later)



Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- **SQL Data Types and Schemas**
- Authorization



Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
 - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
 - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp:** date plus time of day
 - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
 - Example: **interval** '1' day
 - Subtracting a date/time/timestamp value from another gives an interval value
 - Interval values can be added to date/time/timestamp values



Default Values

- **create table student**
*(ID varchar (5),
name varchar (20) not null,
dept_name varchar (20),
tot_cred numeric (3,0) default 0,
primary key (ID))*

```
insert into student(ID,name,dept_name)  
values('12789', 'Newman', 'Comp. Sci.');
```



Index Creation

- **create table student**
*(ID varchar (5),
name varchar (20) not null,
dept_name varchar (20),
tot_cred numeric (3,0) default 0,
primary key (ID))*

create index studentID_index on student(ID)



Index Creation

- Indices are data structures used to speed up access to records with specified values for index attributes

- e.g. **select ***
from student
where ID = '12345'

can be executed by using the index to find the required record, without looking at all records of *student*

More on indices in Chapter 11



Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*.
 - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
 - **clob**: character large object -- object is a large collection of character data

book_review clob(10KB)

image blob(10MB)

movie blob(2GB)



Large-Object Types

- When a query returns a large object, a “locator” is returned rather than the large object itself.
- The locator can then be used to fetch the large object in small pieces, rather than all at once
- Much like reading data from an operating system file using a read function call



User-Defined Types

- SQL supports two forms of user-defined data types:
 - distinct types
 - structured data types
 - allows the creation of complex data types with nested record structures, arrays and multisets (Chapter 22)



User-Defined Types

- **create type** construct in SQL creates user-defined type

```
create type Dollars as numeric (12,2) final
```

- **create table** *department*
(dept_name varchar (20),
building varchar (15),
budget Dollars);
- NOTE: The keyword final isn't really meaningful in this context but is required by theSQL:1999 standard; some implementations allow the final keyword to be omitted.



User-Defined Types

- It is possible for several attributes to have the same data type.
 - e.g., the name attributes for student name and instructor (the set of all person names)
 - but not instructor name and dept_name (we would normally not consider the query “Find all instructors who have the same name as a department”)
 - ⇒ assigning an instructor’s name to a department name is probably a programming error
 - Similarly, comparing a monetary value expressed in dollars and pounds

create type Dollars as numeric (12,2) final
create type Pounds as numeric (12,2) final



User-Defined Types

- Declaring different types for different attributes results to strong type checking
 - e.g., `(department.budget+20)` would not be accepted
 - The attribute and the integer constant 20 have different types
- Solution:
 - Values of one type can be cast (converted) to another domain:

cast (*department.budget* **to** *numeric (12,2)*)



Domains

- **create domain** construct in SQL-92 creates user-defined domain types

```
create domain person_name char(20) not null
```

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree_level* **varchar(10)**
constraint *degree_level_test*
check (value in ('Bachelors', 'Masters', 'Doctorate'));



Create Table Extensions

- Creating tables that have the same schema as an existing table.

```
create table temp_instructor like instructor
```

```
create table t1 as  
  (select *  
   from instructor  
   where dept_name= 'Music')  
with data
```



Create Table Extensions

- **create table ... as** statement closely resembles the create view statement and both are defined by using queries.
- The main difference is that the contents of the table are set when the table is created, whereas the contents of a view always reflect the current query result



Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization



Authorization

- Forms of authorization on parts of the database:
 - **Read** - allows reading, but not modification of data.
 - **Insert** - allows insertion of new data, but not deletion or updating of existing data.
 - **Update** - allows updating, but not insertion or deletion of data.
 - **Delete** - allows deletion of data, but not insertion or updating.
- Each of these authorization types is called a **privilege**
- A user who creates a new relation is given **all privileges** on that relation automatically



Authorization

Forms of authorization to modify the database schema

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.



Authorization Specification in SQL

- The **grant** statement is used to confer authorization

```
grant <privilege list>
on <relation name or view name>
to <user/role list>
```

- <user list> is:
 - a user-id
 - **public**, which allows all valid users the privilege granted
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).



Privileges in SQL

- **select:** allows read access to relation, or the ability to query using the view

- Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *instructor* relation:

grant select on *instructor* to U_1 , U_2 , U_3

- **update:**

grant update on *instructor* to $U1$, $U2$, $U3$



Privileges in SQL

- The authorization may be given either on all attributes of the relation or on only some, but not on specific tuples.
- If the list of attributes is omitted, the privilege will be granted on all attributes of the relation.

grant update on *instructor* to *U1, U2, U3*

grant update (*name*) on *instructor* to *U1, U2, U3*



Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

revoke <privilege list>

on <relation name or view name>

from <user/role list>

- Example:

revoke select on department from U_1, U_2, U_3

revoke update (budget) on department from U_1, U_2, U_3



Roles

- Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users.
- Each database user is granted a set of roles that he/she is authorized to perform.

create role *lecturer*;

grant *lecturer* **to** *U₁*;

grant select on *takes* **to** *lecturer*;



Roles

- Roles can be granted to users, as well as to other roles

create role *teaching_assistant*

grant *teaching_assistant* **to** *lecturer*;

- *lecturer* inherits all privileges of *teaching_assistant*

- Chain of roles

create role *dean*;

grant *instructor* **to** *dean*;

grant *dean* **to** *U₂*;

- When a user logs in to the database system, the actions executed by the user during that session have
 - all the privileges granted directly to the user
 - all privileges granted to roles that are granted (directly or indirectly via other roles) to that user



Authorization on Views

- Authorization on view gives us the possibility to define authorization with respect to some specific tuples

```
create view geo_instructor as
(select *
from instructor
where dept_name = 'Geology');
```

```
grant select on geo_instructor to geo_staff
```

- Then a *geo_staff* member can issue

```
select *
from geo_instructor;
```

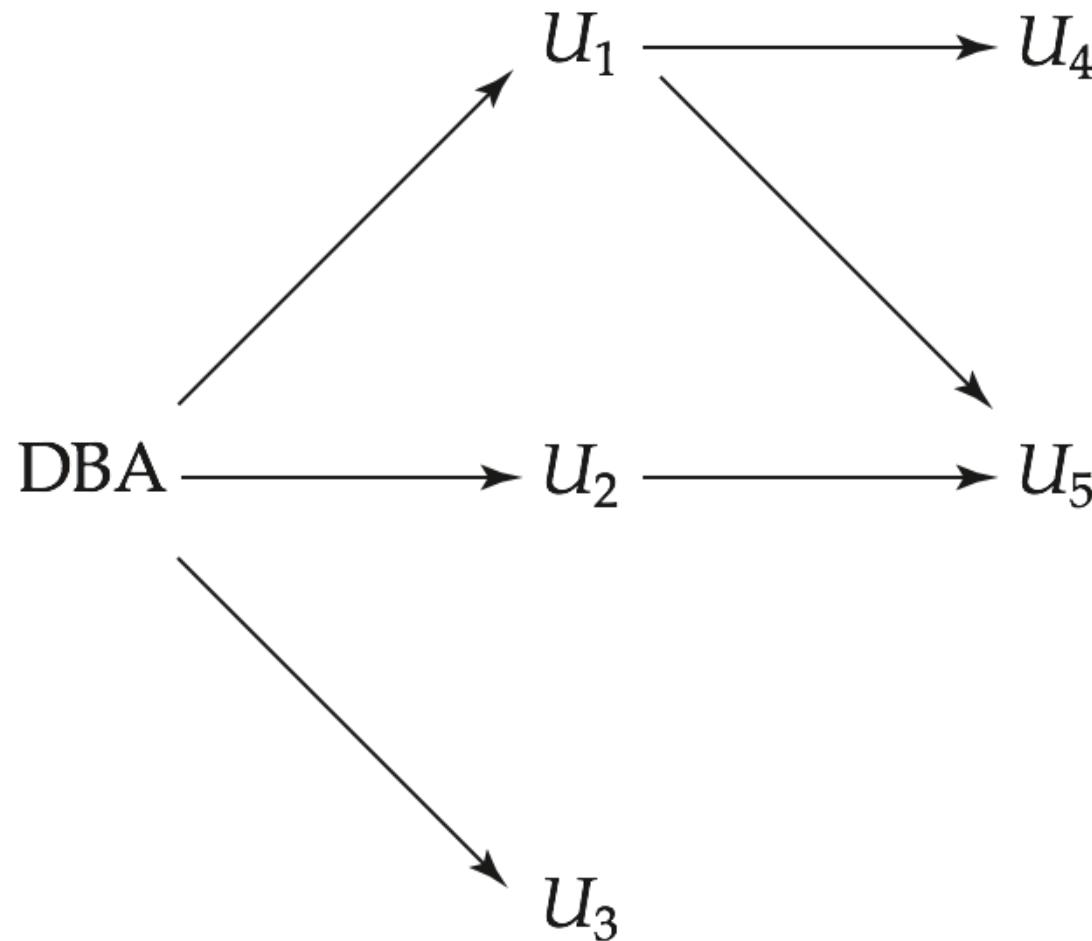


Other Authorization Features

- **references** privilege to create foreign key
 - **grant reference** (*dept_name*) **on** *department* **to** U_1 ;
- transfer of privileges
 - **grant select** **on** *department* **to** U_1 **with grant option**;
 - **revoke select** **on** *department* **from** U_1, U_2 **cascade**;
 - **revoke select** **on** *department* **from** U_1, U_2 **restrict**;



Transfer of privileges





Questions?



Database Systems

Lecture 6: Advanced SQL

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book



Outline

- Accessing SQL From a Programming Language
- Functions
- Triggers
- Advanced Aggregation Features
- OLAP



Accessing SQL From a Programming Language

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
 - Connect with the database server
 - Send SQL commands to the database server
 - Fetch tuples of result one-by-one into program variables



Accessing SQL From a Programming Language

- Possible approaches:
 - Dynamic SQL
 - ▶ JDBC (Java Database Connectivity) works with Java
 - ▶ ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic.
 - Other API's such as ADO.NET sit on top of ODBC
 - Embedded SQL



JDBC

- JDBC is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.



JDBC

- Model for communicating with the database:
 - Open a connection
 - Create a “statement” object
 - Execute queries using the Statement object to send queries and fetch results
 - Exception mechanism to handle errors



JDBC Example

```
public static void JDBCexample(String userid, String passwd)
{
    try
    {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                    "insert into instructor values(77987, 'Kim', 'Physics', 98000)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select dept_name, avg (salary) "+
                " from instructor "+
                " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept.name") + " " +
Apago PDF Enhancer
                rset.getFloat(2));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```



JDBC Example

```
public static void JDBCExample(String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
            userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
    }
```



JDBC Example

```
'ResultSet rset = stmt.executeQuery(  
        "select dept_name, avg (salary) "+  
        " from instructor "+  
        " group by dept_name");  
while (rset.next()) {  
    System.out.println(rset.getString("dept_name") + " " +  
        ApagoPDFEnhancer  
    )}  
stmt.close();  
conn.close();  
}  
catch (Exception sqle)  
{  
    System.out.println("Exception : " + sqle);  
}  
}
```



Database Connection

- Each database product that supports JDBC provides a JDBC driver that must be dynamically loaded in order to access the database from Java.
 - This is done by invoking `Class.forName` with one argument specifying a concrete class implementing the `java.sql.Driver` interface
- Connecting to the Database: A connection is opened using the `getConnection` method of the `DriverManager` class (within `java.sql`) using 3 parameters:
 - a string that specifies the URL, or machine name, where the server runs
 - a database user identifier, which is a string
 - a password, which is also a string.
 - ▶ Note: the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.



Shipping SQL Statements

- Methods for executing a statement:
 - *executeQuery*
 - ▶ When the SQL statement is a query
 - ▶ It returns a result set
 - *executeUpdate*
 - ▶ When the SQL statement is nonquery (DDL or DML)
 - Update
 - Insert
 - Delete
 - Create table
 - ...
 - ▶ It returns an integer giving the number of tuples inserted, updated, or deleted.
 - ▶ For DDL statements, the return value is zero.



Retrieving the Results of a Query

- Retrieving the set of tuples in the result into a ResultSet object
- Fetching the results one tuple at a time
- Using the *next* method on the result set to test whether there remains at least one unfetched tuple in the result set and if so, fetches it.
- Attributes from the fetched tuple are retrieved using various methods whose names begin with *get*
 - *getString*: can retrieve any of the basic SQL data types
 - *getFloat*
- Possible argument to the *get* methods
 - The attribute name specified as a string
 - An integer indicating the position of the desired attribute within the tuple



Database Connection

- The statement and connection are both closed at the end of the Java program.
- It is important to close the connection because there is a limit imposed on the number of connections to the database
- Unclosed connections may cause that limit to be exceeded.
- If this happens, the application cannot open any more connections to the database.



Prepared Statements

- Creating a prepared statement in which some values are replaced by “?”
- Specifying that actual values will be provided later
- Compiling the query by the database system when it is prepared
- Reusing the previously compiled form of the query and apply the new values whenever the query is executed
 - (with new values to replace the “?”s),



Prepared Statements

```
PreparedStatement pStmt = conn.prepareStatement(  
        "insert into instructor values(?, ?, ?, ?);  
pStmt.setString(1, "88877");  
pStmt.setString(2, "Perry");  
pStmt.setString(3, "Finance");  
pStmt.setInt(4, 125000);  
pStmt.executeUpdate();  
pStmt.setString(1, "88878");  
pStmt.executeUpdate();
```

Figure 5.2 Prepared statements in JDBC code.

- insert into instructor values (“88877”, “Perry”, “Finance”, 125000)
- insert into instructor values (“88878”, “Perry”, “Finance”, 125000)



Prepared Statements

- Prepared statements allow for more efficient execution
 - Where the same query can be compiled once and then run multiple times with different parameter values
 - Whenever a user-entered value is used, even if the query is to be run only once



Metadata Features

- Capturing metadata about
 - Database
 - ResultSet (relations)

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```



ODBC

- Open DataBase Connectivity (ODBC) standard
 - standard for application program to communicate with a database server.
 - application program interface (API) to
 - ▶ open a connection with a database,
 - ▶ send queries and updates,
 - ▶ get back results.



Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- A language to which SQL queries are embedded is referred to as a **host language**
- The SQL structures permitted in the host language comprise *embedded SQL*
- An embedded SQL program must be processed by a special **preprocessor prior to compilation.**
- The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow runtime execution of the database accesses.
- Then, the resulting program is compiled by the host-language compiler.



Embedded SQL

- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

```
EXEC SQL <embedded SQL statement>;
```

- Note: this varies by language:
 - In some languages, like COBOL, the semicolon is replaced with END-EXEC
 - In Java embedding uses

```
# SQL { .... };
```



Database Connection

- Before executing any SQL statements, the program must first connect to the database. This is done using:

EXEC-SQL connect to *server* user *user-name* using *password*;

- Here, *server* identifies the server to which a connection is to be established.



Variables

- Variables of the host language can be used within embedded SQL statements.
- They are preceded by a colon (:) to distinguish from SQL variables (e.g., `:credit_amount`)
- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

```
EXEC-SQL BEGIN DECLARE SECTION;
```

```
    int credit-amount ;
```

```
EXEC-SQL END DECLARE SECTION;
```



SQL Query

- To write an embedded SQL query, we use the following statement:

declare c cursor for <SQL query>

- The variable c is used to identify the query
- Example:

EXEC SQL

```
declare c cursor for
    select ID, name
    from student
    where tot_cred > :credit_amount
```

END_EXEC



SQL Query

- The `open` statement is then used to evaluate the query
- The `open` statement for our example is as follows:

```
EXEC SQL open c ;
```

- This statement causes the database system to execute the query and to save the results within a temporary relation.
- The query uses the value of the host-language variable *credit-amount* at the time the `open` statement is executed.



SQL Query

■ The fetch statement

- Placing the values of one tuple in the query result into host language variables
- Requiring one host-language variable for each attribute of the result relation;
 - e.g., we need one variable to hold the ID value (*si*) and another to hold the name value (*sn*) which have been declared within a **DECLARE** section

EXEC SQL

fetch c into :si, :sn

END_EXEC

Repeated calls to fetch get successive tuples in the query result



SQL Query

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c ;
```



Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Example for updating tuples fetched by cursor by declaring that the cursor is for update

EXEC SQL

```
declare c cursor for
    select *
    from instructor
    where dept_name = 'Music'
    for update;
```



Updates Through Embedded SQL

- Iterating through the tuples by performing **fetch** operations on the cursor
- Executing the following statement

EXEC SQL

```
update instructor
      set salary = salary + 1000
      where current of c;
```



Outline

- Accessing SQL From a Programming Language
- **Functions**
- Triggers
- Advanced Aggregation Features
- OLAP



Functions and Procedures

- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
 - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
 - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.



SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count(dept_name varchar(20))
    returns integer
begin
    declare d_count integer;
    select count (*) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```



SQL Functions

- The function `dept_count` can be used in a query that returns names and budget of all departments with more than 12 instructors.

```
select dept_name, budget  
from department  
where dept_count (dept_name ) > 12
```



SQL Functions

- Compound statement: **begin ... end**
 - May contain multiple SQL statements between **begin** and **end**.
- **returns** -- indicates the variable-type that is returned (e.g., integer)
- **return** -- specifies the values that are to be returned as result of invoking the function
- SQL functions are in fact **parameterized views** that generalize the regular notion of views by allowing parameters.



Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))  
returns table (  
    ID varchar(5),  
    name varchar(20),  
    dept_name varchar(20),  
    salary numeric(8,2))  
  
return table  
(select ID, name, dept_name, salary  
from instructor  
where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *  
from table (instructor_of ('Music'))
```



Outline

- Accessing SQL From a Programming Language
- Functions
- **Triggers**
- Advanced Aggregation Features
- OLAP



Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
 - Specify the conditions under which the trigger is to be executed.
 - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
 - Syntax illustrated here may not work exactly on your database system; check the system manuals



Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
 - For example, **after update of takes on grade**
- Values of attributes before and after an update can be referenced
 - **referencing old row as** : for deletes and updates
 - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.



Trigger Example: Using set Statement

```
create trigger setnull before update on takes  
referencing new row as nrow  
for each row  
when (nrow.grade = '')  
begin atomic  
    set nrow.grade = null;  
end;
```



Trigger Example: to Maintain Referential Integrity

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
    select time_slot_id
    from time_slot) /* time_slot_id not present in time_slot */
begin
    rollback
end;

create trigger timeslot_check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
    select time_slot_id
    from time_slot) /* last tuple for time_slot_id deleted from time_slot */
and orow.time_slot_id in (
    select time_slot_id
    from section)) /* and time_slot_id still referenced from section*/
begin
    rollback
end;
```



Trigger Example: to Maintain credits_earned value

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
    and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred+
        (select credits
         from course
         where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```



Disabling Triggers

- Triggers can be disabled or enabled;
 - by default they are enabled when they are created.
- Triggers can be disabled by:
`alter trigger trigger_name disable`
`disable trigger trigger_name`
- A trigger that has been disabled can be enabled again.
- A trigger can instead be dropped, which removes it permanently, by:
`drop trigger trigger_name`



When Not To Use Triggers

- Triggers were used earlier for tasks such as
 - Maintaining summary data (e.g., total salary of each department)
 - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
 - Databases today provide built in materialized view facilities to maintain summary data
 - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
 - Define methods to update fields
 - Carry out actions as part of the update methods instead of through a trigger



When Not To Use Triggers

- Risk of unintended execution of triggers, for example, when
 - Loading data from a backup copy
 - Replicating updates at a remote site
 - Trigger execution can be disabled before such actions.
- Other risks with triggers:
 - Error leading to failure of critical transactions that set off the trigger
 - Cascading execution



Outline

- Accessing SQL From a Programming Language
- Functions
- Triggers
- **Advanced Aggregation Features**
- OLAP



Ranking

- Suppose we are given a relation
student_grades(*ID*, *GPA*)
giving the grade-point average of each student
- Goal: finding the rank of each student.
- Ranking can be done using basic SQL aggregation, but
resultant query is very inefficient

```
select ID, (1 + (select count(*)  
                 from student_grades B  
                 where B.GPA > A.GPA)) as s_rank  
from student_grades A  
order by s_rank;
```



Ranking

- Ranking is done in conjunction with an order by specification.

```
select ID, rank() over (order by GPA desc) as s_rank
  from student_grades
        order by s_rank
```

- NOTE: the extra **order by** clause is needed to get them in sorted order



Ranking

- Two possible approaches for ranking
 - Leaving gaps: (default)
e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
 - Without gaps: (using **dense_rank**)
so next dense rank would be 2



Ranking with Partitions

- Ranking can be done within partition of the data.
- “Find the rank of students within each department.”

```
select ID, dept_name,
      rank () over (partition by dept_name order by GPA desc)
          as dept_rank
from dept_grades
order by dept_name, dept_rank,
```

- Ranking is done *after* applying **group by** clause/aggregation



Top n Items

- Can be used to find top-n results
 - More general than the **limit n** clause supported by many databases, since it allows top-n within each partition



Other Ranking Functions

■ **ntile:**

- For a given constant n , the ranking function $ntile(n)$ takes the tuples in each partition in the specified order, and divides them into n buckets with equal numbers of tuples.
- E.g.,

```
select ID, ntile(4) over (order by GPA desc) as quartile  
from student_grades;
```



Other Ranking Functions

- SQL:1999 permits the user to specify **nulls first** or **nulls last**

```
select ID,  
       rank ( ) over (order by GPA desc nulls last) as s_rank  
from student_grades
```



Outline

- Accessing SQL From a Programming Language
- Functions
- Triggers
- Advanced Aggregation Features
- OLAP



Data Analysis and OLAP

■ Online Analytical Processing (OLAP)

- Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Is used for **multidimensional data** (data that can be modeled as dimension attributes and measure attributes)
 - **Measure attributes**
 - ▶ measure some value
 - ▶ can be aggregated upon
 - ▶ e.g., the attribute *number* of the *sales* relation
 - **Dimension attributes**
 - ▶ define the dimensions on which measure attributes (or aggregates thereof) are viewed
 - ▶ e.g., attributes *item_name*, *color*, and *size* of the *sales* relation



Example sales relation

item_name	color	clothes_size	quantity
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	2
...
...



Cross Tabulation of sales by *item_name* and *color*

clothes_size

all

<i>item_name</i>	<i>color</i>				total
	dark	pastel	white		
skirt	8	35	10		53
dress	20	10	5		35
shirt	14	7	28		49
pants	20	2	5		27
total	62	54	48		164

- The table above is an example of a **cross-tabulation (cross-tab)**, also referred to as a **pivot-table**.
 - Values for one of the dimension attributes form the row headers
 - Values for another dimension attribute form the column headers
 - Other dimension attributes are listed on top
 - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.



Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab
- Can have n dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube

		item_name					clothes_size			
		skirt	dress	shirt	pants	all	small	medium	large	all
color		2	5	3	1	11	16	18	45	77
dark	2	8	5	7	22	4	34	9	42	77
	8	20	14	20	62	16	4	18	45	77
pastel	35	10	7	2	54	21	42	9	45	77
	10	8	28	5	48	34	42	18	45	77
white	53	38	49	27	164	42	45	18	45	77
	all	all	all	all	all	all	all	all	all	all



Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab
 - Each cross-tab is a two-dimensional view on a multidimensional data cube.
 - With an OLAP system, a data analyst can look at different cross-tabs on the same data by interactively selecting the attributes in the cross-tab.
 - Examples:
 - ▶ selecting a cross-tab on item name and clothes size
 - ▶ selecting a cross-tab on color and clothes size



Online Analytical Processing Operations

■ **Slicing:** creating a cross-tab for fixed values only

- OLAP systems allow an analyst to see a cross-tab on item name and color for a fixed value of clothes size,
- Example:
 - ▶ large, instead of the sum across all sizes.
- This operation is referred to as slicing, since it can be thought of as viewing a slice of the data cube.
- The operation is sometimes called **dicing**.



Cross Tabulation With Hierarchy

- The following table can be achieved using natural join with another table specifying item_names and category

clothes_size: **all**

<i>category</i>	<i>item_name</i>	<i>color</i>			
		dark	pastel	white	total
womenswear	skirt	8	8	10	53
	dress	20	20	5	35
	subtotal	28	28	15	88
menswear	pants	14	14	28	49
	shirt	20	20	5	27
	subtotal	34	34	33	76
total		62	62	48	164



Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
- Can drill down or roll up on a hierarchy
 - roll up: moving from finer granularity to coarser-granularity (by the mean of aggregation)
 - drill down: moving from coarser-granularity to finer granularity (must be generated from original data)



Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations

- The value **all** is used to represent aggregates.
- The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

item_name	color	clothes_size	quantity
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	White	all	28
shirt	all	all	49
pant	dark	all	20
pant	pastel	all	2
pant	white	all	5
pant	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164



Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
- Example relation for this section
sales(item_name, color, clothes_size, quantity)
- E.g. consider the query

```
select item_name, color, size, sum(number)
from sales
group by cube(item_name, color, size)
```

This computes the union of eight different groupings of the *sales* relation:

```
{ (item_name, color, size),
  (item_name, color),  (item_name, size),  (color, size),
  (item_name),        (color),            (size),          () }
```

where () denotes an empty **group by** list.

- For each grouping, the result contains the null value for attributes not present in the grouping.



Extended Aggregation

- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,

```
select item_name, color, size, sum(number)  
from sales  
group by rollup(item_name, color, size)
```

Generates union of four groupings:

```
{ (item_name, color, size),  
(item_name, color),  
(item_name),  
( ) }
```



Extended Aggregation

- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory(item_name, category)* gives the category of each item. Then

```
select category, item_name, sum(number)
from sales, itemcategory
where sales.item_name = itemcategory.item_name
group by rollup(category, item_name)
```

would give a hierarchical summary by *item_name* and by *category*.



Extended Aggregation

- Multiple rollups or cubes can be used in a single group by clause
 - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

```
select item_name, color, size, sum(number)
from sales
group by rollup(item_name), rollup(color, size)
```

generates the groupings

$$\begin{aligned} & \{item_name, ()\} \times \{(color, size), (color), ()\} \\ &= \{ (item_name, color, size), (item_name, color), (item_name), \\ & \quad (color, size), (color), () \} \end{aligned}$$



OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.
- OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.



OLAP Implementation

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
 - Space and time requirements for doing so can be very high
 - ▶ 2^n combinations of **group by**
- Several optimizations available for computing multiple aggregates
 - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
 - ▶ Can compute aggregate on $(item_name, color)$ from an aggregate on $(item_name, color, size)$
 - is cheaper than computing it from scratch



Questions?



Database System

Lecture 7: Entity-Relationship Model

Dr. Momtazi momtazi@aut.ac.ir

Based on the slides of the course book



Outline

- **Design Process**
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling



Database Design

- The task of creating a database application involves
 - Design of the database schema
 - Design of the programs that access and update the data
 - Design of a security scheme to control access to data



Design Phases

- Characterizing the data needs of the prospective database users
- Choosing a data model
- By applying the concepts of the chosen data model, translating the requirements into a conceptual schema of the database
- Reviewing the schema to ensure it meets functional requirements



Design Phases

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- Logical Design – Deciding on the database schema.
Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database



Design Approaches

- Entity Relationship Model (current chapter)
 - Models an enterprise as a collection of *entities* and *relationships*
 - ▶ Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - Described by a set of *attributes*
 - ▶ Relationship: an association among several entities
 - Represented diagrammatically by an *entity-relationship diagram*:
- Normalization Theory (Chapter 8)
 - Formalize what designs are bad, and test for them



Outline

- Design Process
- **E-R Data Modeling**
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling



ER model -- Database Modeling

- The ER data model was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the ER model.



Major issues

- In designing a database schema, we must ensure that we avoid two major pitfalls:

- **Redundancy**: A bad design may repeat information.

Example: if we store the course identifier and title of a course with each course offering, the title would be stored redundantly with each course offering. It would suffice to store only the course identifier with each course offering, and to associate the title with the course identifier only once, in a course entity

- **Incompleteness**: A bad design may make certain aspects of the enterprise difficult or impossible to model.

Example: if we only have entities corresponding to sections, without having an entity corresponding to courses and repeat all of the course information once for each section, it would then be impossible to represent information about a new course, unless that course is offered.



ER model -- Database Modeling

- The ER data model employs three basic concepts:
 - entity sets
 - relationship sets
 - attributes
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a database graphically.



Entity

- An **entity** is an object that exists and is distinguishable from other objects.
 - Example: instructors, students, departments, courses

- An **entity set** is a set of entities of the same type that share the same properties.
 - Example: set of all instructors, students, departments, courses



Attributes

- An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.

- Example:

instructor = (ID, name, street, city, salary)

course= (course_id, title, credits)

- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.



Entity Sets -- *instructor* and *student*

instructor_ID instructor_name

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

student-ID student_name

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student



Relationship

- A **relationship** is an association among several entities

Example: An instructor advising a student, a student taking an offered course, an instructor teaching an offered course

44553 (Peltier)
student entity

advisor
relationship set

22222 (Einstein)
instructor entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

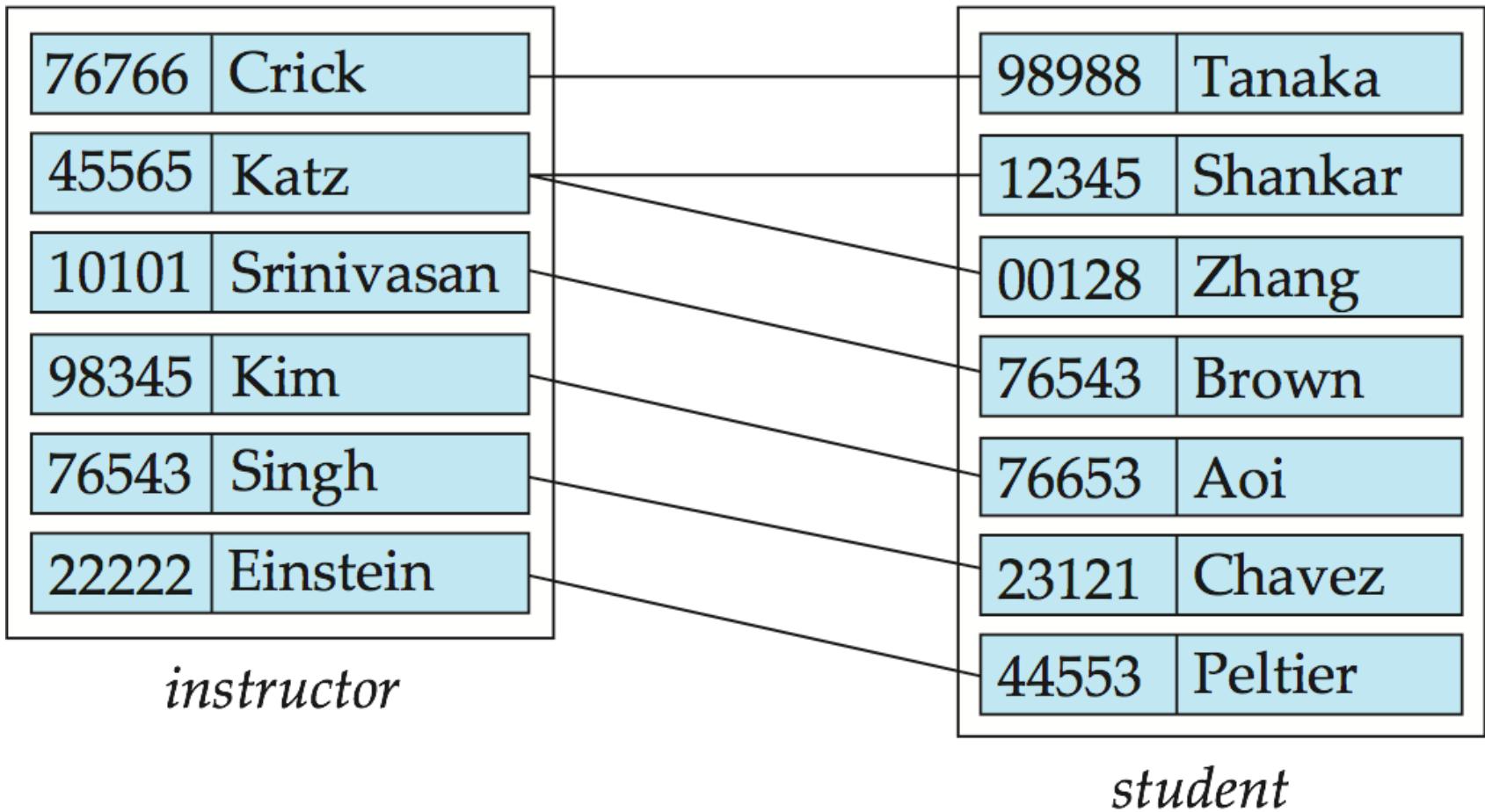
where (e_1, e_2, \dots, e_n) is a relationship

- Example:

$$(44553, 22222) \in \text{advisor}$$



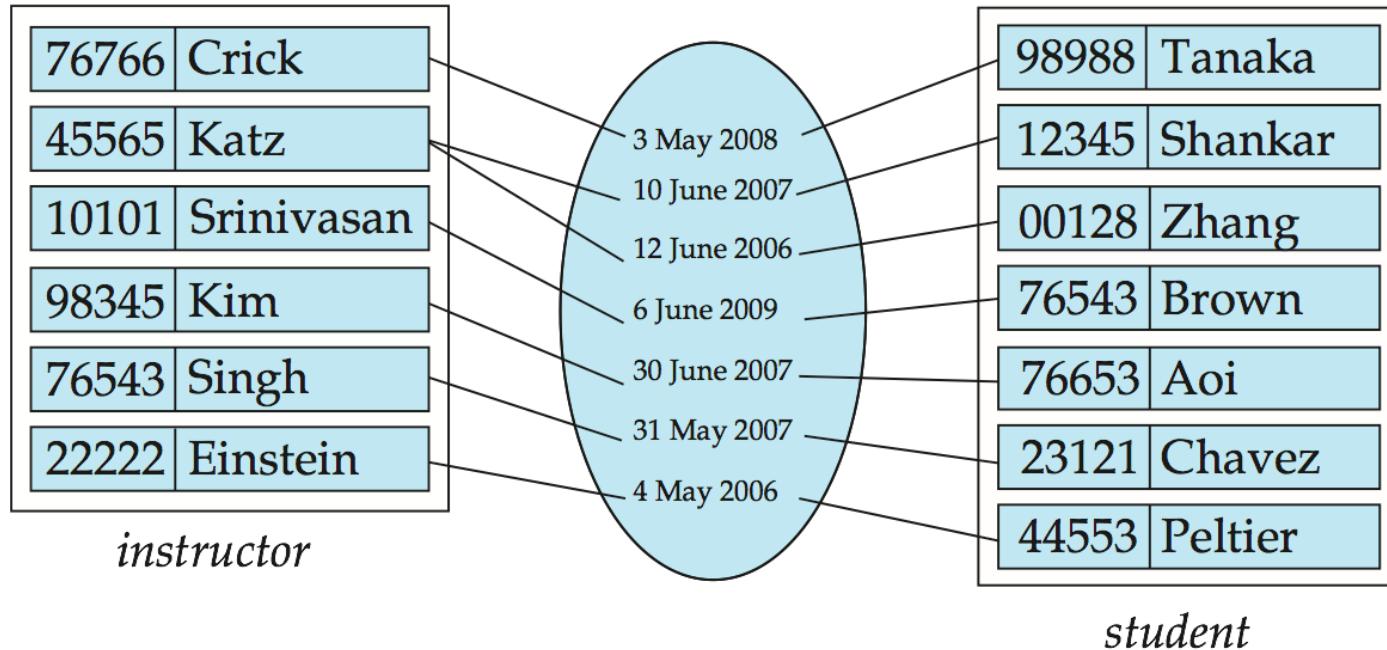
Relationship Set *advisor*





Descriptive Attributes

- An attribute can also be associated with a relationship set.
- Example: the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor





Degree of a Relationship Set

- Binary relationship
 - involve two entity sets (or degree two).
 - most relationship sets in a database system are binary.
 - Example: Advisor relationship between instructor and student

- Relationships involving more than two entity sets
 - Example: *students* work on research *projects* under the guidance of an *instructor*.
 - relationship *proj_guide* is a ternary relationship between *instructor*, *student*, and *project*



Outline

- Design Process
- E-R Data Modeling
- **E-R Constraints**
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling

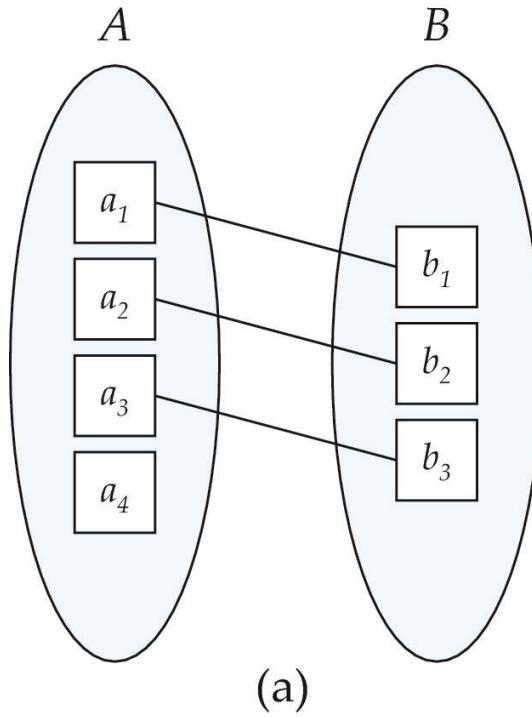


Mapping Cardinality Constraints

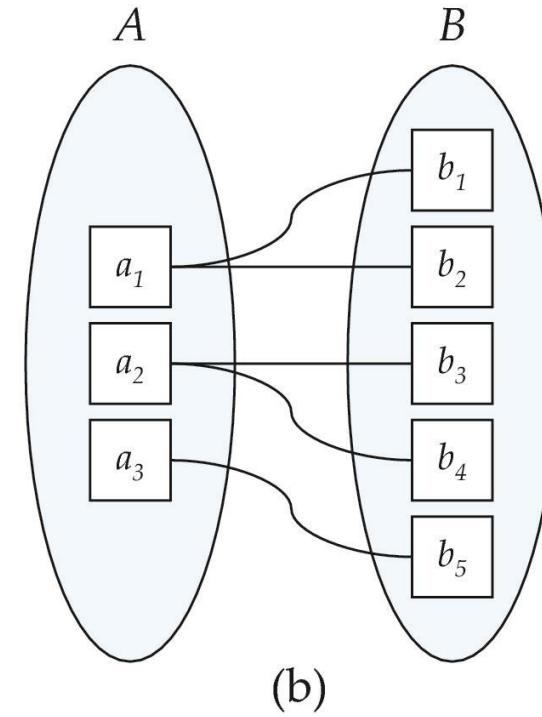
- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many



Mapping Cardinalities



One to one

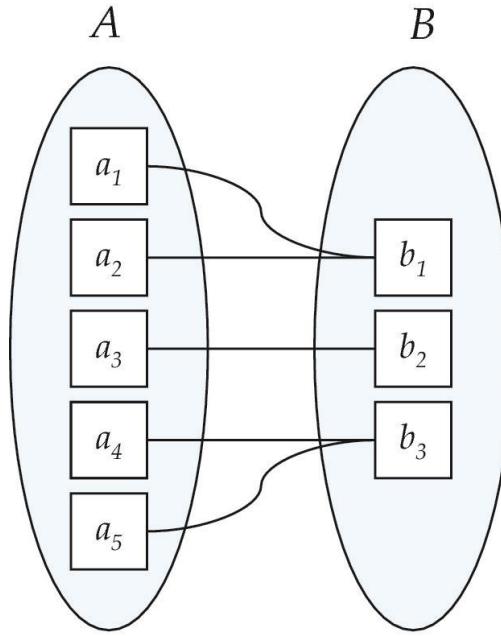


One to many

Note: Some elements in A and B may not be mapped to any elements in the other set

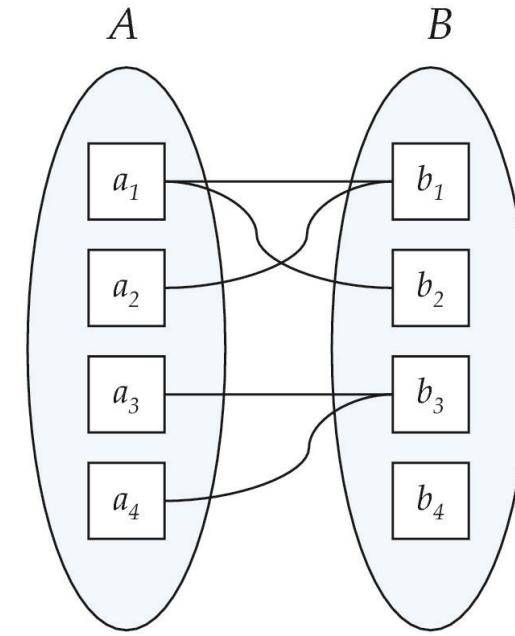


Mapping Cardinalities



(a)

Many to
one



(b)

Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set



Total and Partial Participation

- Total participation: every entity in the entity set participates in at least one relationship in the relationship set
 - Example: participation of *student* in *advisor* relation is total
 - ▶ every *student* must have an associated instructor
- Partial participation: some entities may not participate in any relationship in the relationship set
 - Example: participation of *instructor* in *advisor* is partial
 - *An instructor need not advise any students*
 - *It is possible that only some of the instructor entities are related to the student entity set through the advisor relationship*



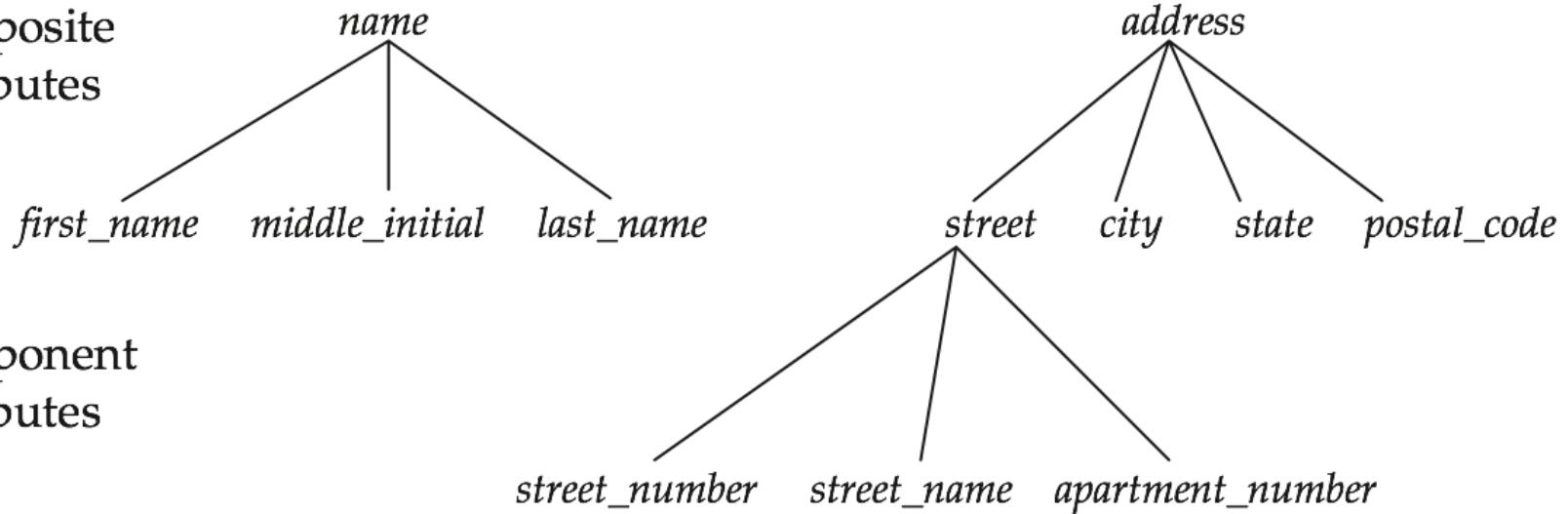
Complex Attributes

- Attribute types:
 - **Simple** and **composite** attributes.
 - **Single-valued** and **multivalued** attributes
 - ▶ Example: multivalued attribute: *phone_numbers*
 - **Derived** attributes
 - ▶ Can be computed from other attributes
 - ▶ Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute



Composite Attributes

composite
attributes



component
attributes



Designing Attributes

- Once the entity sets are decided upon, we must choose the appropriate attributes.
- These attributes are supposed to represent the various values we want to capture in the database.
- The choice of what attributes to include is up to the designer, who has a good understanding of the structure of the enterprise.

- Example: for the instructor entity set,
 - we included the attributes ID, name, dept name, and salary.
 - We can also add the attributes phone number, office number, home_page, etc.



Outline

- Design Process
- E-R Data Modeling
- E-R Constraints
- **Removing Redundancy**
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling



Redundant Attributes

- Suppose we have entity sets: *instructor* and *department*
 - We should model the fact that each instructor has an associated department
 - Two possible approaches:
 - Using the attribute *dept_name* appears in *instructor*.
 - ▶ *instructor*: *ID, name, dept_name, salary*
 - ▶ *department*: *dept_name, building, budget*
 - Using a relationship set *inst_dept*
 - ▶ *instructor*: *ID, name, salary*
 - ▶ *department*: *dept_name, building, budget*
 - ▶ *inst_dept*: *ID, dept_name*
- (the attribute *dept_name* in *instructor* is redundant and should be removed)



Weak Entity Sets

- Suppose we want to model the fact that each course is offered within one or more sections
- Two possible approaches:
 - Using the attribute *course_id* appears in *section*
 - Using the relationship set *sec_course* between entity sets *section* and *course*
 - ▶ *course_id* is redundant information appeared in *sec_course*, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
 - ▶ One option to deal with this redundancy is to not store the attribute *course_id* in the section entity and to only store the remaining attributes *section_id*, year, and semester. However, the entity set *section* then does not have enough attributes to identify a particular section entity uniquely; different courses may share the same *section_id*, year, and semester.



Weak Entity Sets

- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**;
 - Example:
 - ▶ weak entity: *section*
 - ▶ identifying entity: *course*
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.
- An entity set that is not a weak entity set is termed a **strong entity set**.



Weak Entity Sets

- Every weak entity must be associated with an identifying entity
 - The weak entity set is said to be **existence dependent** on the identifying entity set.
 - The identifying entity set is said to **own** the weak entity set that it identifies.
 - The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.



List of Entity Sets and their Attributes

- **classroom**: with attributes (building, room_number, capacity).
- **department**: with attributes (dept_name, building, budget).
- **course**: with attributes (course_id, title, credits).
- **instructor**: with attributes (ID, name, salary).
- **section**: with attributes (course_id, sec_id, semester, year).
- **student**: with attributes (ID, name, tot_cred).
- **time_slot**: with attributes (time_slot_id, {(day, start_time, end_time)}).



List of Relationships

- `inst_dept`: relating instructors with departments.
- `stud_dept`: relating students with departments.
- `teaches`: relating instructors with sections.
- `takes`: relating students with sections, with a descriptive attribute *grade*.
- `course_dept`: relating courses with departments.
- `sec_course`: relating sections with courses.
- `sec_class`: relating sections with classrooms.
- `sec_time_slot`: relating sections with time slots.
- `advisor`: relating students with instructors.
- `prereq`: relating courses with prerequisite courses.



Outline

- Design Process
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- **E-R Diagram**
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling



Entity Sets

- Entities can be represented graphically as follows:
 - Rectangles represent entity sets.
 - Attributes listed inside entity rectangle
 - Underline indicates primary key attributes

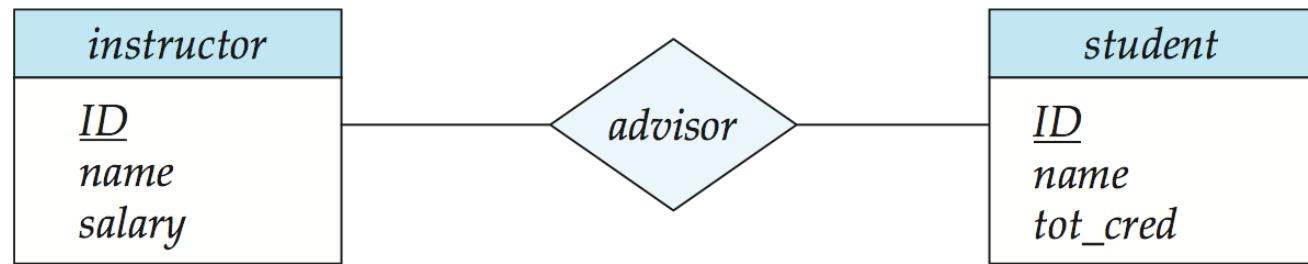
<i>instructor</i>
<u>ID</u>
<i>name</i>
<i>salary</i>

<i>student</i>
<u>ID</u>
<i>name</i>
<i>tot_cred</i>



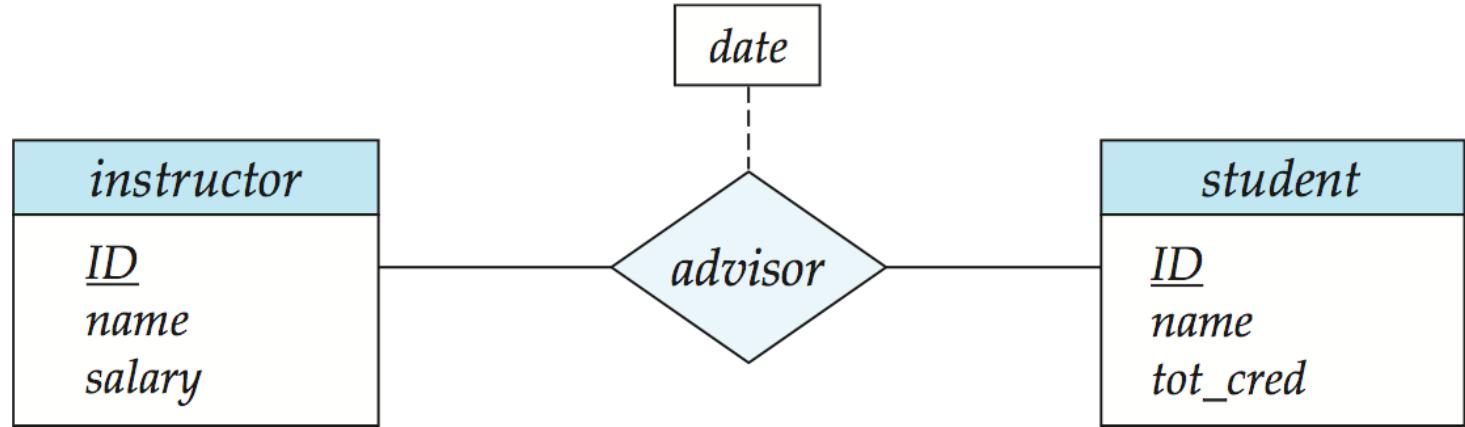
Relationship Sets

- Diamonds represent relationship sets.





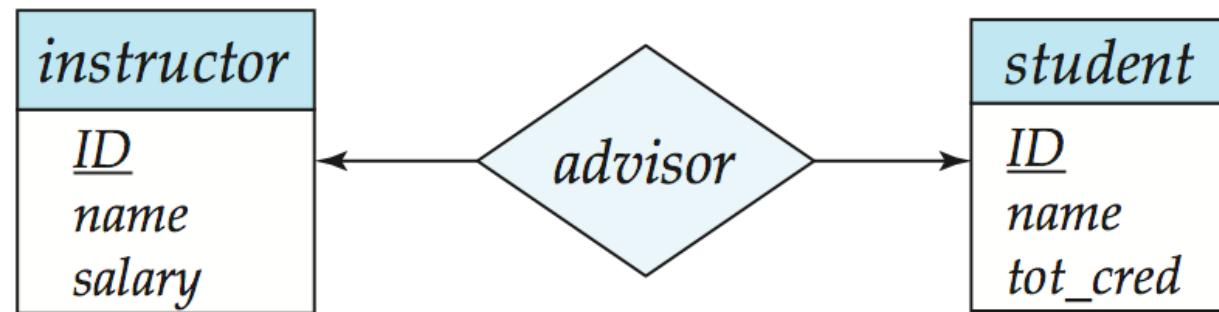
Relationship Sets with Attributes





Cardinality Constraints

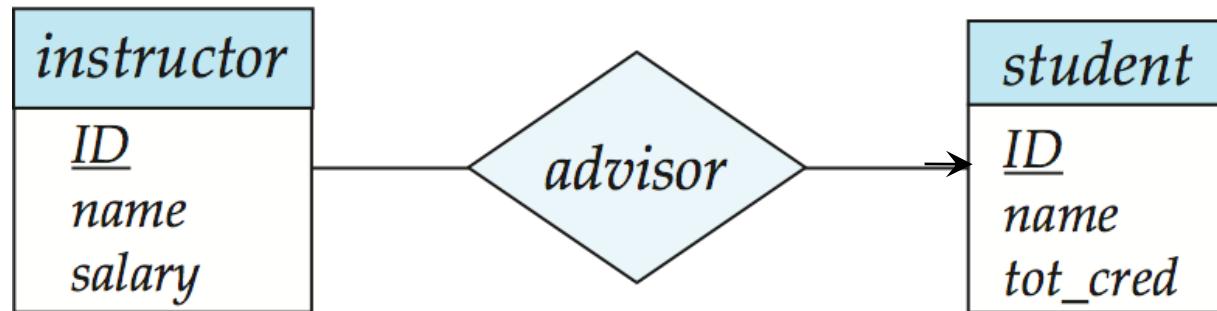
- The cardinality constraints are expressed by drawing either
 - a directed line (\rightarrow), signifying “one” or
 - an undirected line ($-$), signifying “many”
- One-to-one relationship between an *instructor* and a *student*:
 - A student is associated with at most one *instructor* via the relationship *advisor*
 - An *instructor* is associated with at most one *student* via *advisor*





Many-to-One Relationships

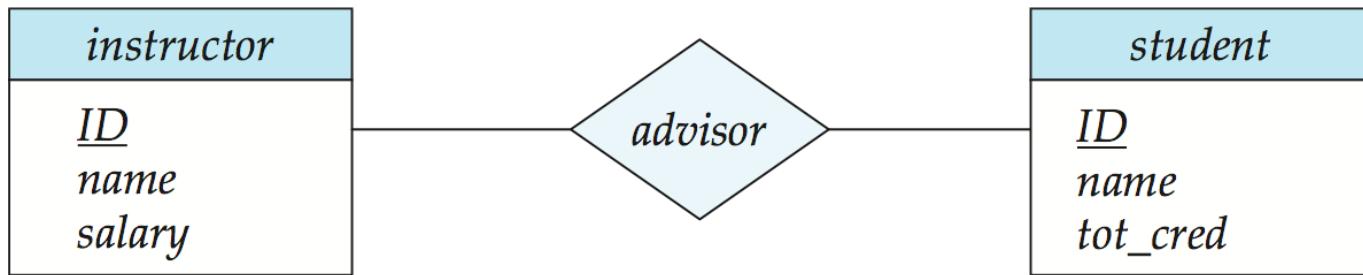
- many-to-one relationship between an *instructor* and a *student*:
 - an *instructor* is associated with at most one *student* via *advisor*
 - a *student* is associated with several (including 0) *instructors* via *advisor*





Many-to-Many Relationship

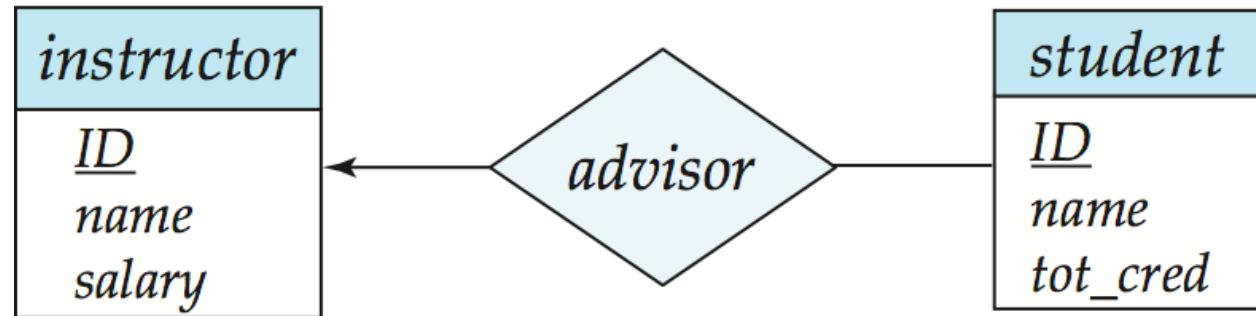
- many-to-many relationship between an *instructor* and a *student*:
 - An instructor is associated with several (possibly 0) students via *advisor*
 - A student is associated with several (possibly 0) instructors via *advisor*





One-to-Many Relationship

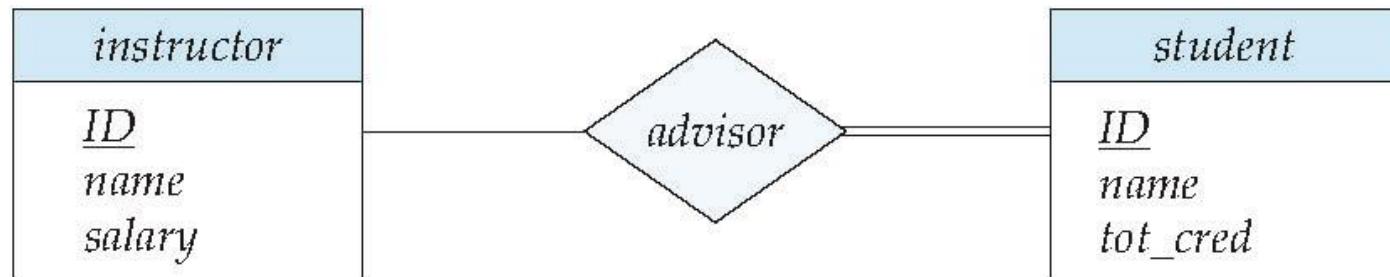
- one-to-many relationship between an *instructor* and a *student*
 - an instructor is associated with several (including 0) students via *advisor* (an instructor may advise many students)
 - a student is associated with at most one instructor via *advisor* (a student may have at most one advisor)





Total and Partial Participation

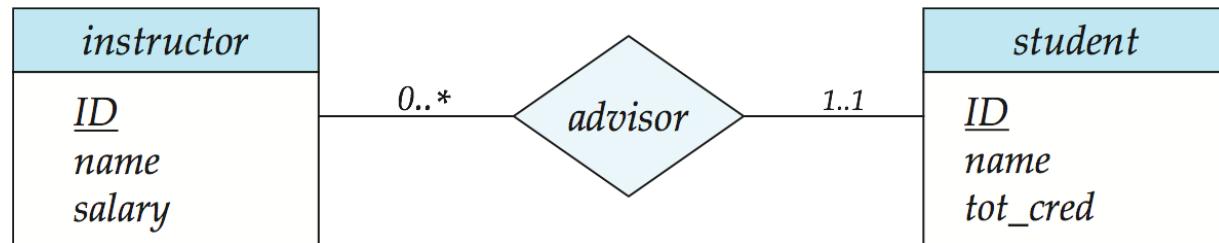
- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
 - ▶ every *student* must have an associated *instructor*
- Partial participation: some entities may not participate in any relationship in the relationship set
 - participation of *instructor* in *advisor* is partial





Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form $l..h$, where l is the minimum and h the maximum cardinality
 - A minimum value of 1 indicates total participation.
 - A maximum value of 1 indicates that the entity participates in at most one relationship
 - A maximum value of * indicates no limit.



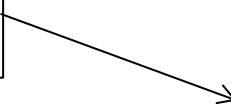
Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors



Notation to Express Entity with Complex Attributes

<i>instructor</i>	
<u>ID</u>	
<i>name</i>	
	<i>first_name</i>
	<i>middle_initial</i>
	<i>last_name</i>
<i>address</i>	
	<i>street</i>
	<i>street_number</i>
	<i>street_name</i>
	<i>apt_number</i>
	<i>city</i>
	<i>state</i>
	<i>zip</i>
	{ <i>phone_number</i> }
	<i>date_of_birth</i>
	<i>age ()</i>

Multi-valued Attribute



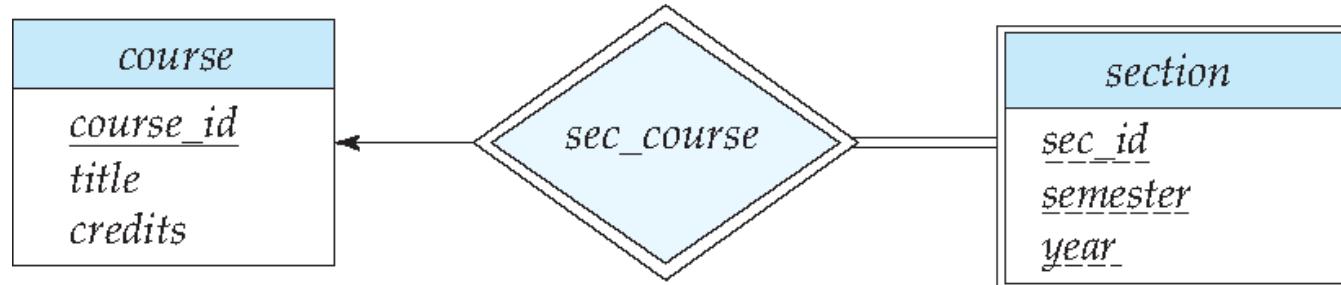
Derived Attribute





Expressing Weak Entity Sets

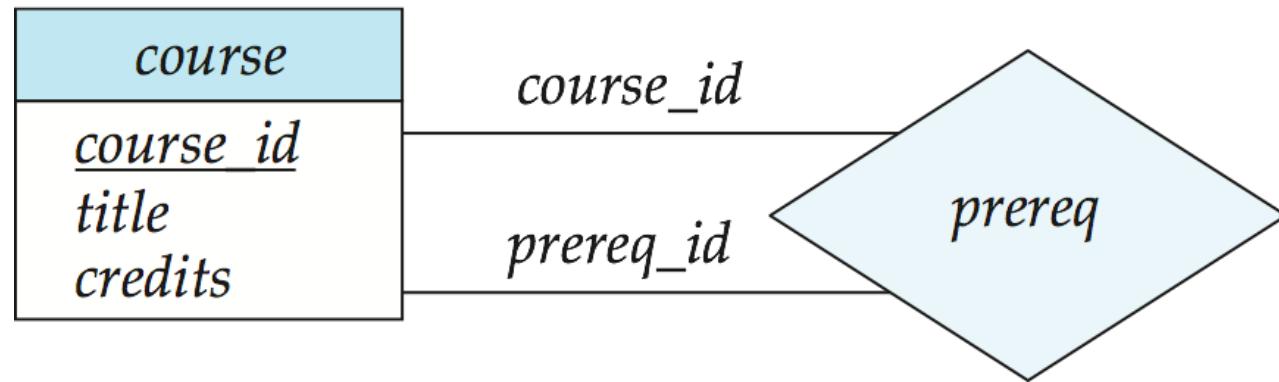
- A weak entity set is depicted via a double rectangle.
- The discriminator of a weak entity set is underlined with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id*, *sec_id*, *semester*, *year*)





Roles

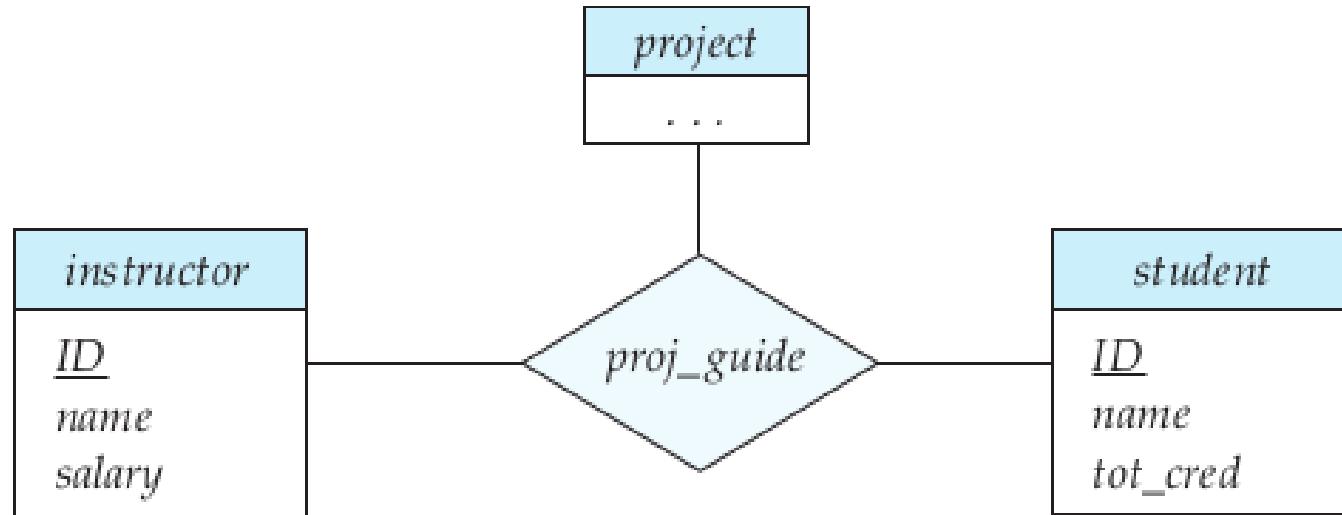
- Entity sets of a relationship need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.





Non-binary Relationship Sets

- three entity sets *instructor*, *student*, and *project*, related through the relationship set *proj_guide*.



NOTE: arrows out of a nonbinary relationship set can be interpreted in different ways.

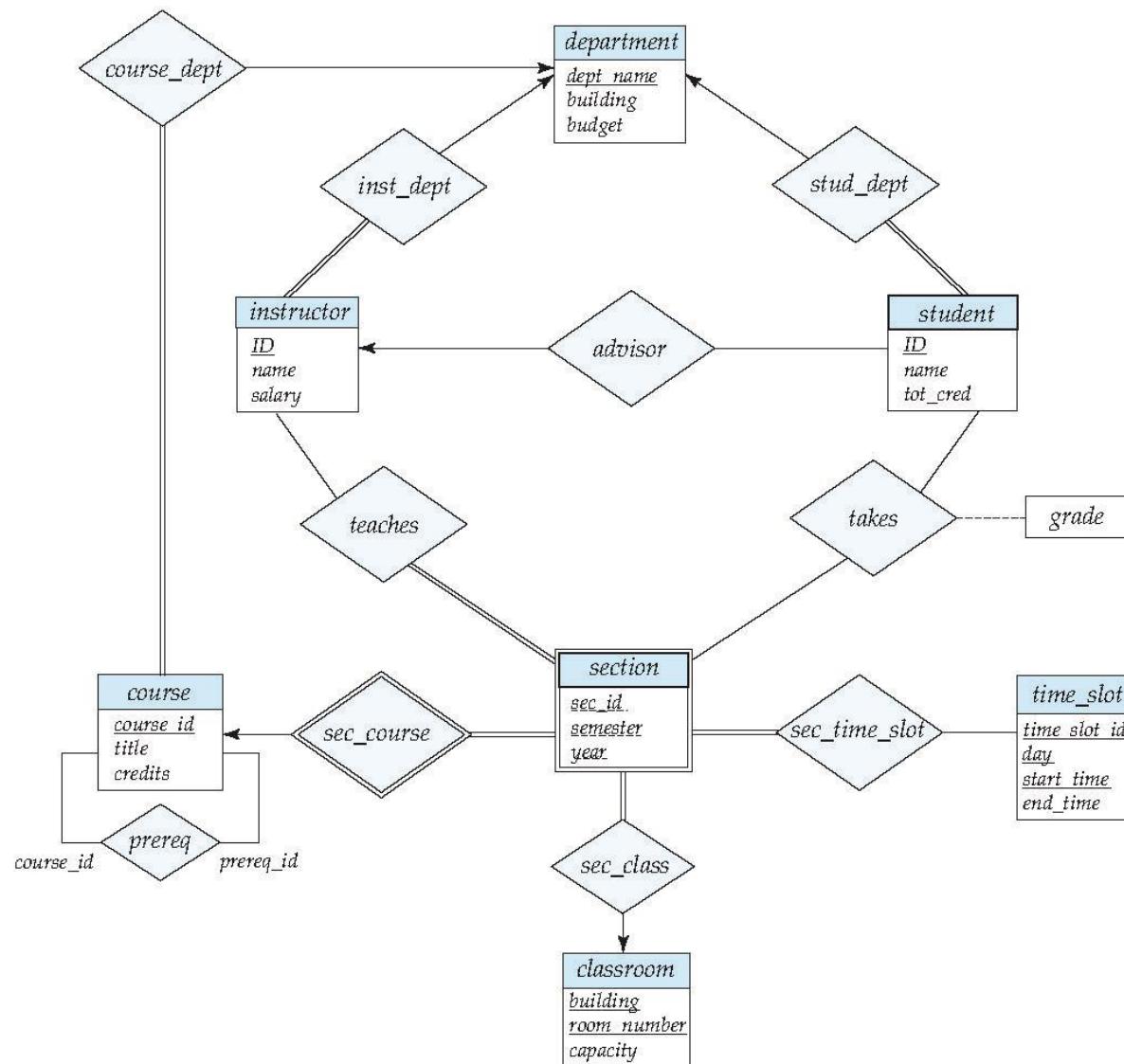


Cardinality Constraints on Ternary Relationship

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- For example, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
 - For example, a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean
 1. Each *A* entity is associated with a unique entity from *B* and *C* or
 2. Each pair of entities from (*A*, *B*) is associated with a unique *C* entity, and each pair (*A*, *C*) is associated with a unique *B*
 - Each alternative has been used in different formalisms
 - To avoid confusion we outlaw more than one arrow



E-R Diagram for a University Enterprise





Outline

- Design Process
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- **Reduction to Relation Schemas**
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling



Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of relation schemas.
- For each entity set and relationship set there is a unique relation schema that is assigned the name of the corresponding entity set or relationship set.
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.



Representing Entity Sets

- A strong entity set reduces to a relation schema with the same attributes

student(ID, name, tot_cred)

<i>student</i>	
<u><i>ID</i></u>	
	<i>name</i>
	<i>tot_cred</i>



Representing Entity Sets

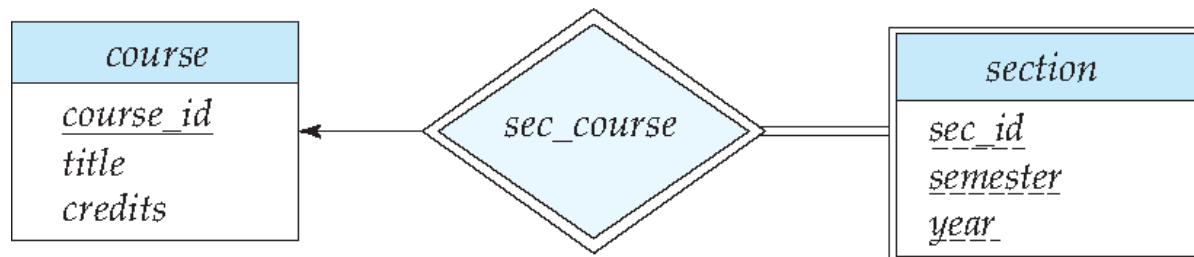
classroom (building, room_number, capacity)
department (dept_name, building, budget)
course (course_id, title, credits)
instructor (ID, name, salary)
student (ID, name, tot_cred)



Representing Entity Sets

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

section (course_id, sec_id, sem, year)





Representation of Entity Sets with Composite Attributes

<i>instructor</i>
<i>ID</i>
<i>name</i>
<i>first_name</i>
<i>middle_initial</i>
<i>last_name</i>
<i>address</i>
<i>street</i>
<i>street_number</i>
<i>street_name</i>
<i>apt_number</i>
<i>city</i>
<i>state</i>
<i>zip</i>
{ <i>phone_number</i> }
<i>date_of_birth</i>
<i>age ()</i>

- Composite attributes are flattened out by creating a separate attribute for each component attribute
 - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
 - ▶ Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name*)
- Ignoring multivalued attributes, extended instructor schema is
 - *instructor(ID, first_name, middle_initial, last_name, street_number, street_name, apt_number, city, state, zip_code, date_of_birth)*



Representation of Entity Sets with Multivalued Attributes

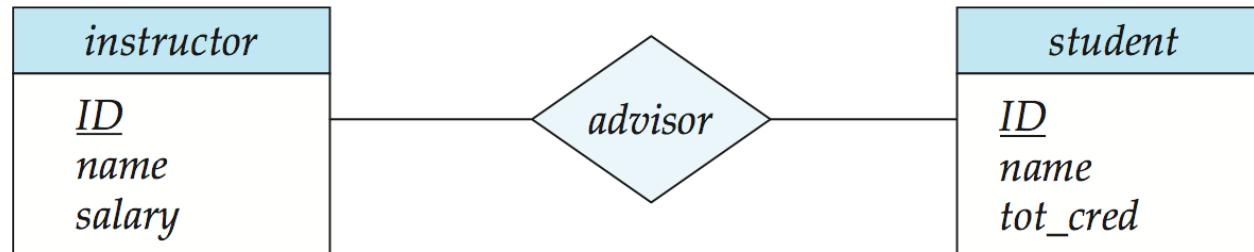
- A multivalued attribute M of an entity E is represented by a separate schema EM
- Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
- Example: Multivalued attribute $phone_number$ of $instructor$ is represented by a schema:
 $inst_phone = (\underline{ID}, \underline{phone_number})$
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema EM
 - For example, an $instructor$ entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples: (22222, 456-7890) and (22222, 123-4567)



Representing Relationship Sets

- A binary many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*

advisor = (s_id, i_id)





Representing Relationship Sets

- For a binary one-to-one relationship set, the primary key of either entity set can be chosen as the primary key. The choice can be made arbitrarily.
- For a binary many-to-one or one-to-many relationship set, the primary key of the entity set on the “many” side of the relationship set serves as the primary key.
- For an n-ary relationship set without any arrows on its edges, the union of the primary key-attributes from the participating entity sets becomes the primary key.
- For an n-ary relationship set with an arrow on one of its edges, the primary keys of the entity sets not on the “arrow” side of the relationship set serve as the primary key for the schema. Recall that we allowed only one arrow out of a relationship set.



Representing Relationship Sets

teaches (ID, course_id, sec_id, semester, year)

takes (ID, course_id, sec_id, semester, year, grade)

prereq (course_id, prereq_id)

advisor (s_ID, i_ID)

sec_course (course_id, sec_id, semester, year)

sec_time_slot (course_id, sec_id, semester, year, time_slot_id)

sec_class (course_id, sec_id, semester, year, building, room_number)

inst_dept (ID, dept_name)

stud_dept (ID, dept_name)

course_dept (course_id, dept_name)



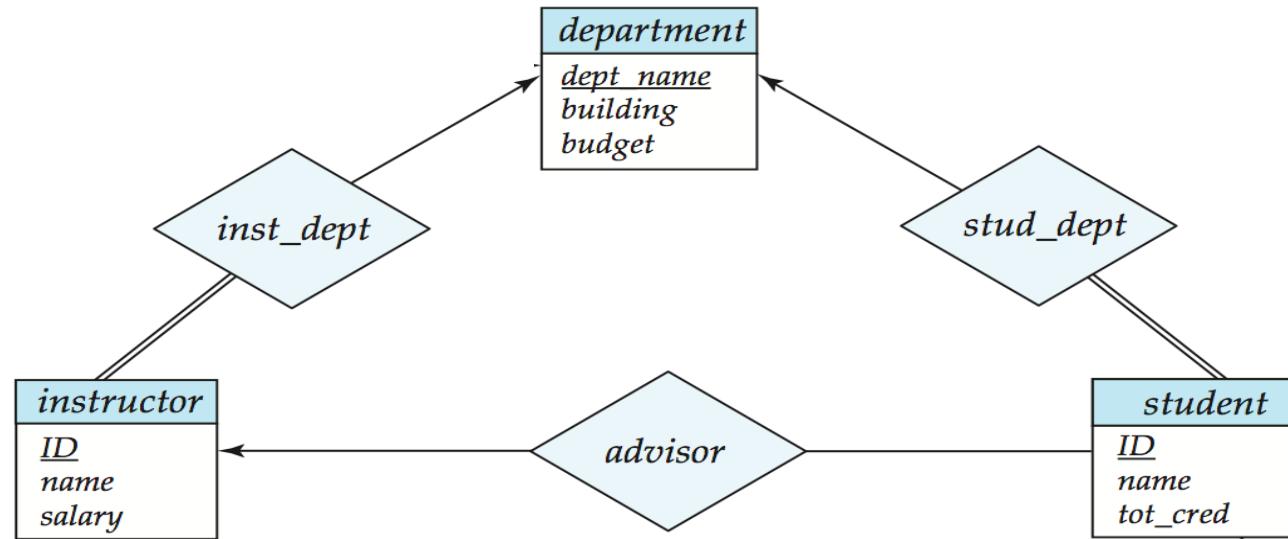
Outline

- Design Process
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- **Redundancy of Schemas**
- Extended E-R Features
- Design Issues
- Alternative Notions of Data Modeling



Redundancy of Schemas

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side
- Example: Instead of creating a schema for relationship set *inst_dept* or *stud_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor* or *student*





Redundancy of Schemas

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
 - That is, an extra attribute can be added to either of the tables corresponding to the two entity sets

- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values



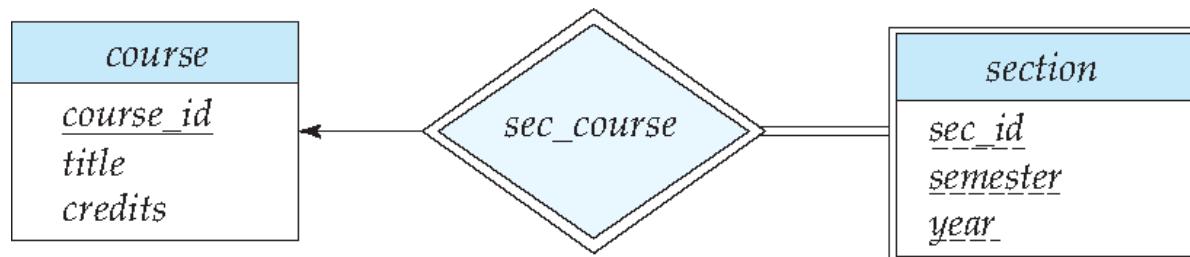
Redundancy of Schemas

- The redundant relations in the E-R diagram:
 - *inst_dept*
 $\Rightarrow \text{instructor: } \{ID, name, dept_name, salary\}$
 - *stud_dept*.
 $\Rightarrow \text{student: } \{ ID, name, dept_name, tot_cred \}$
 - *course_dept*.
 $\Rightarrow \text{course: } \{\text{course_id, title, dept_name, credits}\}$
 - *sec_class*
 $\Rightarrow \text{section: } \{\text{course_id, sec_id, semester, year, building, room_number}\}.$
 - *sec_time_slot*
 $\Rightarrow \text{section: } \{\text{course_id, sec_id, semester, year, building, room_number, time_slot_id}\}$



Redundancy of Schemas

- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
- Example: The *section* schema already contains the attributes that would appear in the *sec_course* schema





Outline

- Design Process
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- **Extended E-R Features**
- Design Issues
- Alternative Notions of Data Modeling



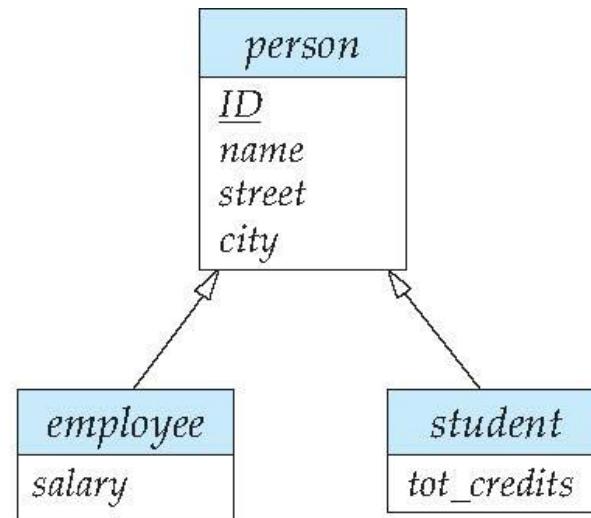
Specialization

- An entity set may include subgroupings of entities that are distinct from other entities in the set. (e.g., some attributes that are not shared by all the entities in the entity set.)
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- The process of designating subgroupings within an entity set is called specialization.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.
- Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**,



Specialization Example

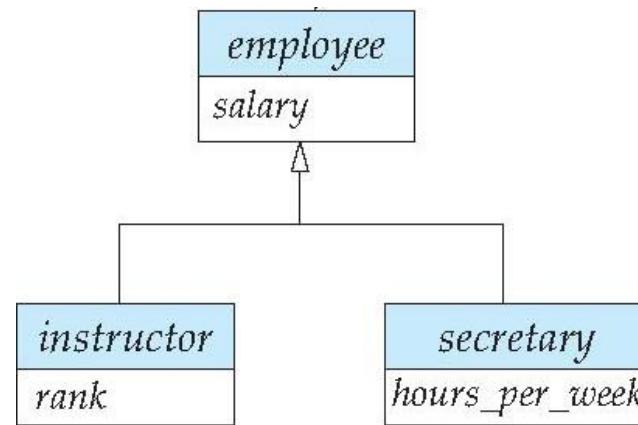
- the entity set person may be further classified as:
 - employee
 - *student*





Specialization Example

- university employees may be further classified as:
 - instructor
 - secretary





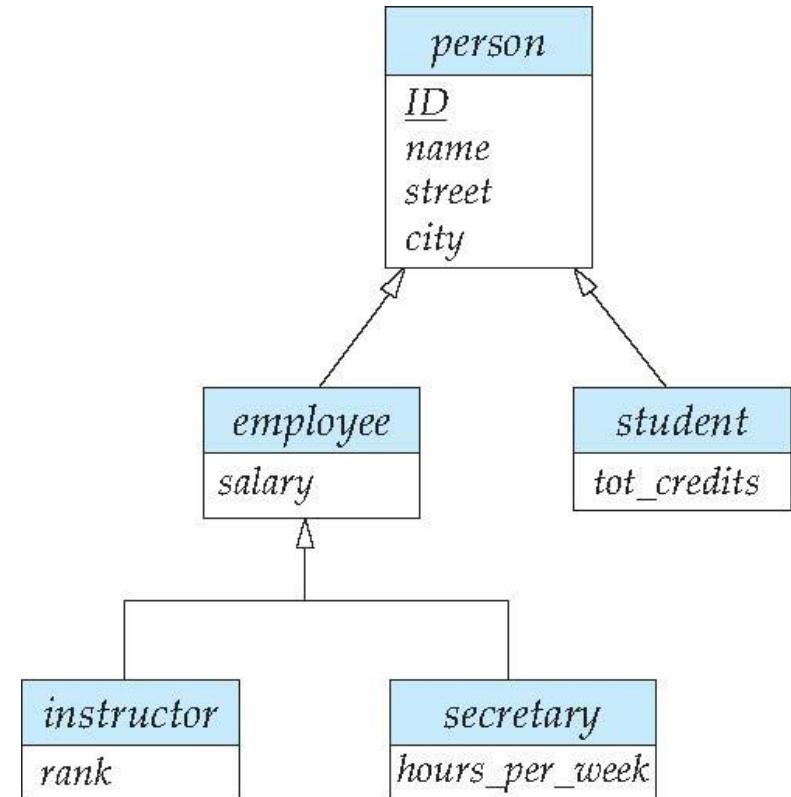
Specialization Example

■ Overlapping

- An entity may belong to multiple specialized entity sets
- Example: *employee* and *student*

■ Disjoint

- Entities must belong to at most one specialized entity set
- *instructor* and *secretary*





Representing Specialization via Schemas

■ Method 1:

- Form a schema for the higher-level entity
- Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

<u>schema</u>	<u>attributes</u>
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

- Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema



Representing Specialization as Schemas

■ Method 2:

- Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

- Drawback: *name*, *street* and *city* may be stored redundantly for people who are both students and employees



Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.
- Differences in the two approaches may be characterized by their starting point and overall goal.
- Whether a given portion of an E-R model was arrived at by specialization or generalization, the outcome is basically the same



Attribute Inheritance

- A crucial property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**.
- The attributes of the higher-level entity sets are inherited by the lower-level entity sets.
- Attribute inheritance applies through all tiers of lower-level entity sets; thus, *instructor* and *secretary*, which are subclasses of *employee*, inherit all the attributes from *person*, in addition to inheriting attributes from *employee*.



Relationship Inheritance

- A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or superclass) participates.
- Like attribute inheritance, participation inheritance applies through all tiers of lower-level entity sets.
 - Example: if the *person* entity set participates in a relationship *person_dept* with *department*. Then, the *student*, *employee*, *instructor* and *secretary* entity sets, which are subclasses of the *person* entity set, also implicitly participate in the *person_dept* relationship with *department*. The above entity sets can participate in any relationships in which the *person* entity set participates.



Design Constraints on a Specialization/Generalization

- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
 - **total**: an entity must belong to one of the lower-level entity sets
 - **partial**: an entity need not belong to one of the lower-level entity sets



Design Constraints on a Specialization/Generalization

- Partial generalization is the default.
- Total generalization can be specified by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The completeness constraint for a generalized higher-level entity set is usually total. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets.
 - Example: the *student* generalization is total; i.e., all student entities must be either graduate or undergraduate.



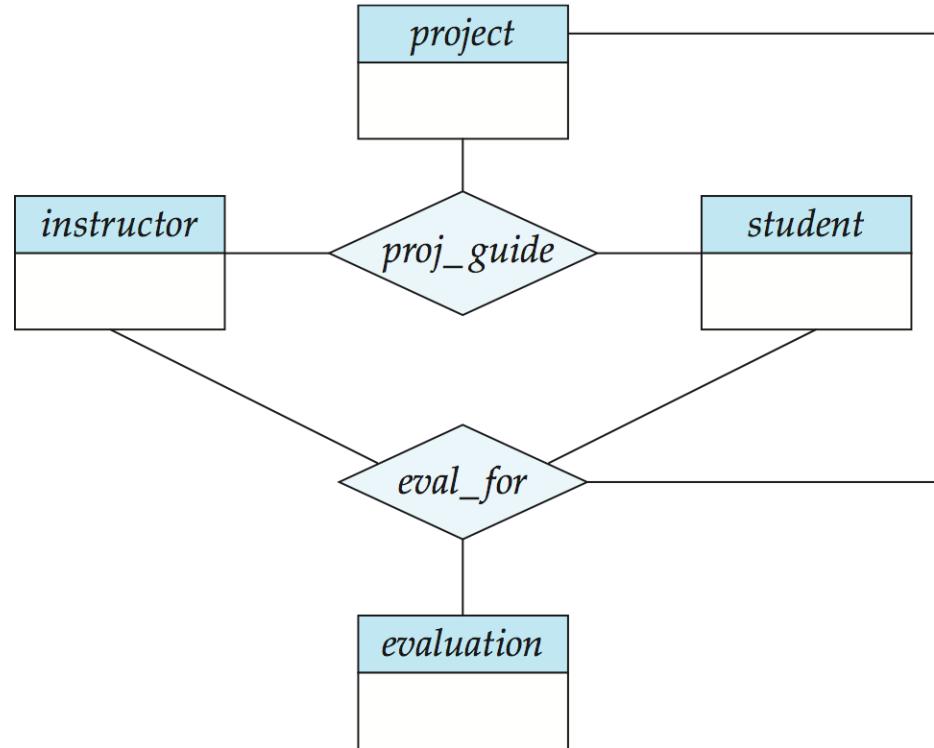
Design Constraints on a Specialization/Generalization

- Certain insertion and deletion requirements follow from the constraints that apply to a given generalization or specialization.
- When a total completeness constraint is in place, an entity inserted into a higher-level entity set must also be inserted into at least one of the lower-level entity sets.
- An entity that is deleted from a higher-level entity set also is deleted from all the associated lower-level entity sets to which it belongs.



Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record monthly evaluation reports of a student by a guide on a project





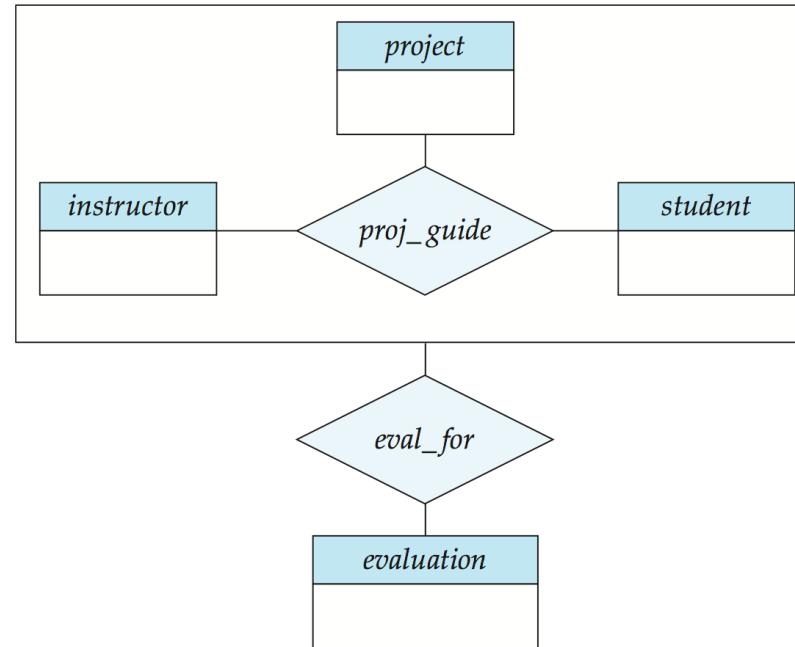
Aggregation

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
 - Every *eval_for* relationship corresponds to a *proj_guide* relationship
 - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
 - ▶ So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
 - Treat relationship as an abstract entity
 - Allows relationships between relationships
 - Abstraction of relationship into new entity



Aggregation

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
 - A student is guided by a particular instructor on a particular project
 - A student, instructor, project combination may have an associated evaluation





Representing Aggregation via Schemas

- To represent aggregation, create a schema containing
 - Primary key of the aggregated relationship
(No separate relation is required to represent the aggregation; the relation created from the defining relationship is used instead.)
 - The primary key of the associated entity set
 - Any descriptive attributes

- In our example:
 - The schema *evaluation* is:
$$\text{evaluation} (\text{evaluation_id}, \text{report})$$
 - The schema *eval_for* is:
$$\text{eval_for} (\text{s_ID}, \text{project_id}, \text{i_ID}, \text{evaluation_id})$$



Outline

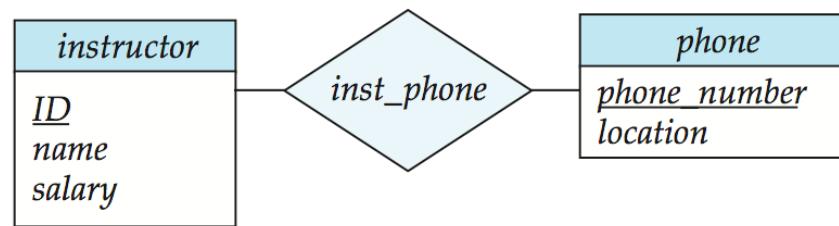
- Design Process
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- **Design Issues**
- Alternative Notions of Data Modeling



Entities vs. Attributes

- Use of entity sets vs. attributes

<i>instructor</i>
<i>ID</i>
<i>name</i>
<i>salary</i>
<i>phone_number</i>

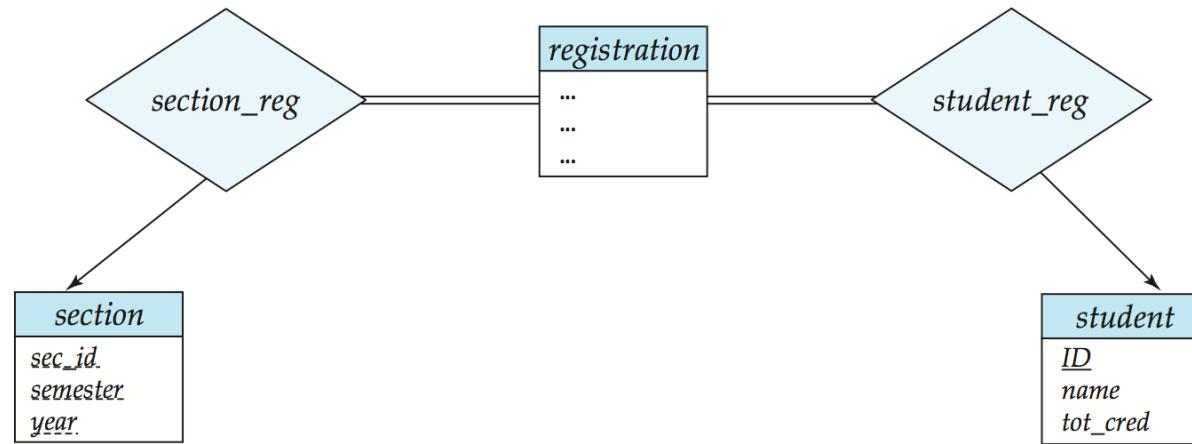


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)



Entities vs. Relationship sets

- Use of entity sets vs. relationship sets



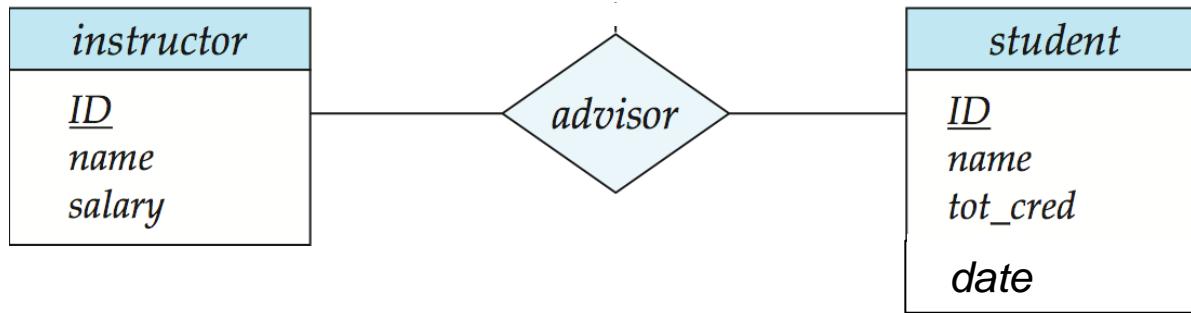
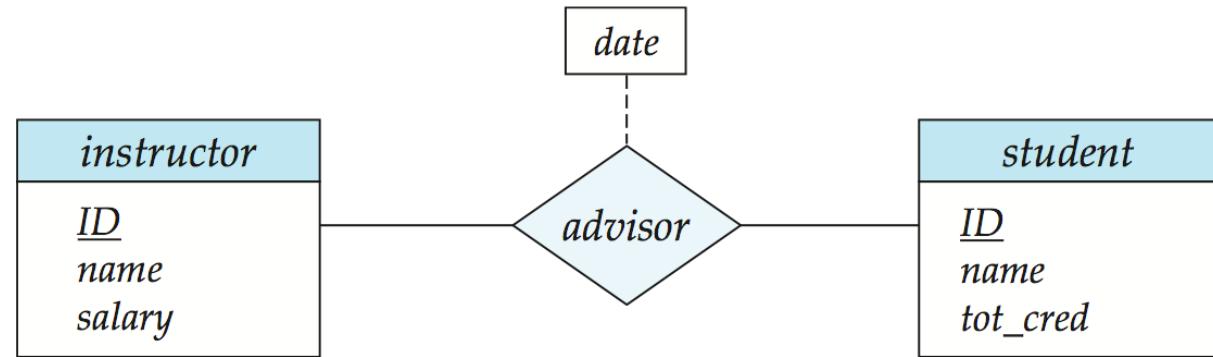
- Possible guideline is to designate a relationship set to describe an action that occurs between entities



Placement of Relationship Attributes

■ Placement of relationship attributes

- Example: attribute date as attribute of advisor or as attribute of student?





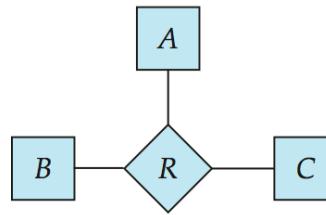
Binary Vs. Non-Binary Relationships

- Although it is possible to replace any non-binary (n -ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a n -ary relationship set shows more clearly that several entities participate in a single relationship.
- Some relationships that appear to be non-binary may be better represented using binary relationships
 - For example, a ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
 - ▶ Using two binary relationships allows partial information
 - But there are some relationships that are naturally non-binary
 - ▶ Example: *proj_guide*

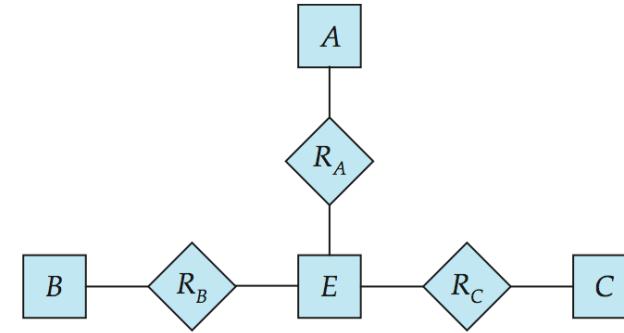


Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
 - Replace R between entity sets A, B and C by an entity set E , and three relationship sets:
 1. R_A , relating E and A
 2. R_B , relating E and B
 3. R_C , relating E and C
 - Create an identifying attribute for E and add any attributes of R to E
 - For each relationship (a_i, b_i, c_i) in R , create
 1. a new entity e_i in the entity set E
 2. add (e_i, a_i) to R_A
 3. add (e_i, b_i) to R_B
 4. add (e_i, c_i) to R_C



(a)



(b)



Converting Non-Binary Relationships

- In case no artificial entity is used for this representation, there may be instances in the translated schema that cannot correspond to any instance of R
 - *Example: we can record that instructor K works on projects A and B with students X and Y; however, we cannot record that K works on project A with student X and works on project B with student Y, but does not work on project A with Y or on project B with X.*



E-R Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

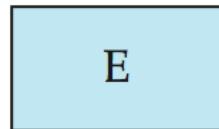


Outline

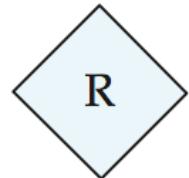
- Design Process
- E-R Data Modeling
- E-R Constraints
- Removing Redundancy
- E-R Diagram
- Reduction to Relation Schemas
- Redundancy of Schemas
- Extended E-R Features
- Design Issues
- **Alternative Notions of Data Modeling**



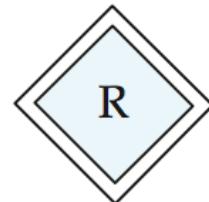
Summary of Symbols Used in E-R Notation



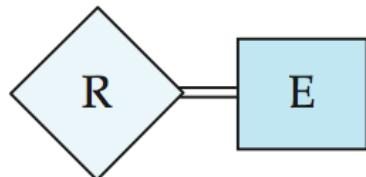
entity set



relationship set



identifying
relationship set
for weak entity set



total participation
of entity set in
relationship

E
A1
A2
A2.1
A2.2
{A3}
A4 ⁰

attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)

E
<u>A1</u>

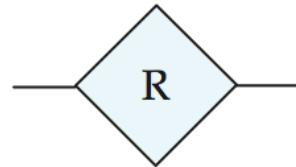
primary key

E
A1
.....

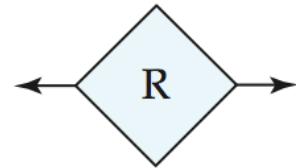
discriminating
attribute of
weak entity set



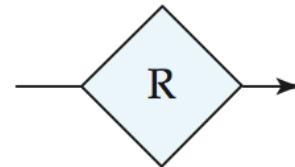
Summary of Symbols Used in E-R Notation



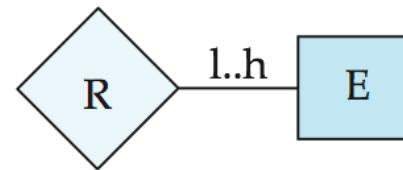
many-to-many
relationship



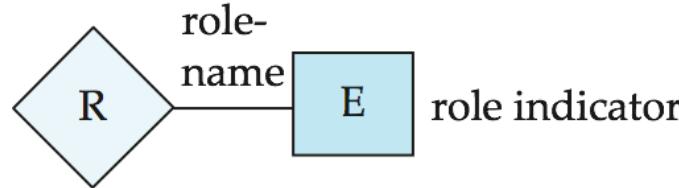
one-to-one
relationship



many-to-one
relationship



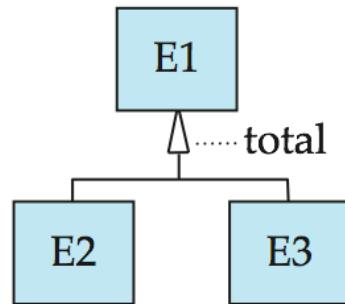
cardinality
limits



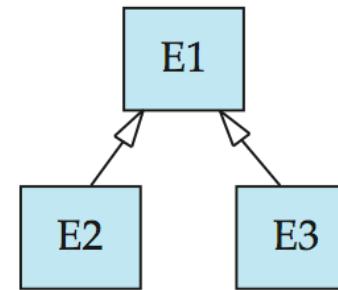
role
indicator



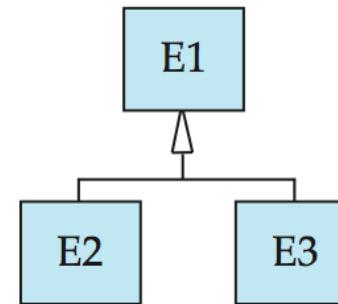
Summary of Symbols Used in E-R Notation



total (disjoint)
generalization



ISA: generalization
or specialization

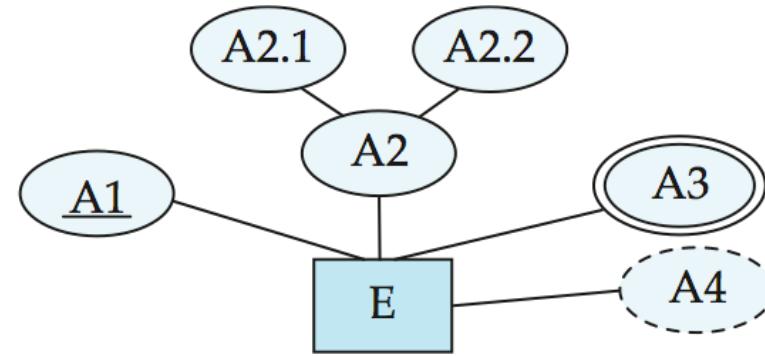


disjoint
generalization

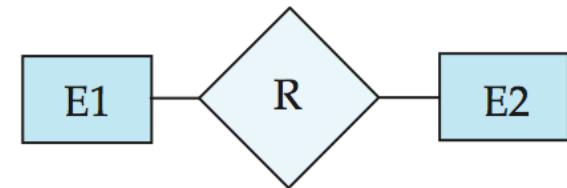


Alternative ER Notations

entity set E with
simple attribute A1,
composite attribute A2,
multivalued attribute A3,
derived attribute A4,
and primary key A1



- Relationship attributes can be similarly represented, by connecting the ovals to the diamond representing the relationship.



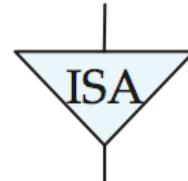


Alternative ER Notations

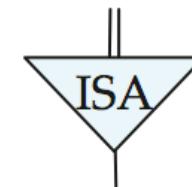
weak entity set



generalization



total
generalization

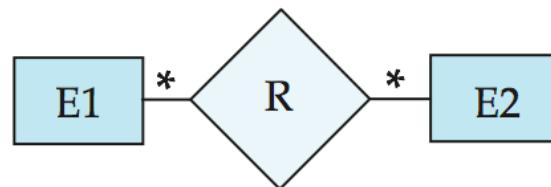




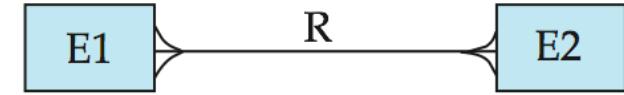
Alternative ER Notations

Chen

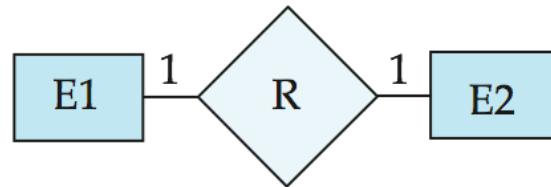
many-to-many
relationship



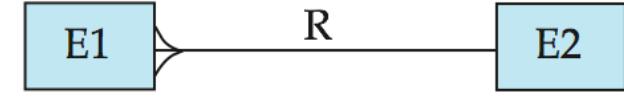
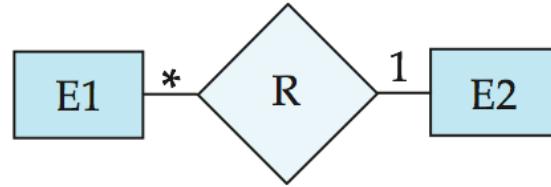
IDE1FX (Crows feet notation)



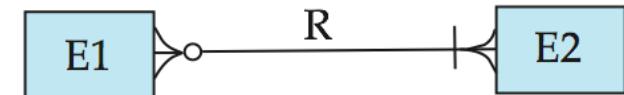
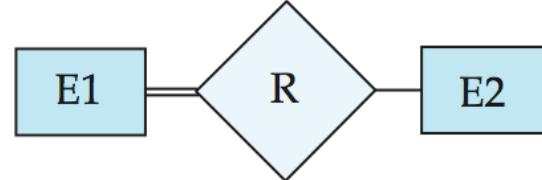
one-to-one
relationship



many-to-one
relationship



participation
in R: total (E1)
and partial (E2)





UML

- **UML:** Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
 - Class diagram.
 - ▶ is similar to an E-R diagram, but with differences
 - Use case diagram
 - Activity diagram
 - Implementation diagram



ER vs. UML Class Diagrams

ER Diagram Notation

E
A1
M1()

entity with
attributes (simple,
composite,
multivalued, derived)

Equivalent in UML

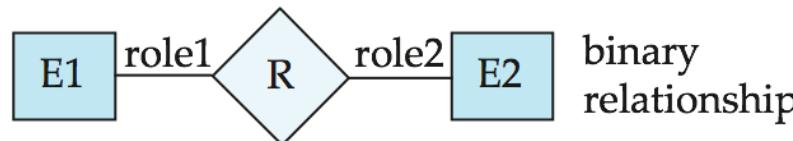
E
-A1
+M1()

class with simple attributes
and methods (attribute
prefixes: + = public,
- = private, # = protected)

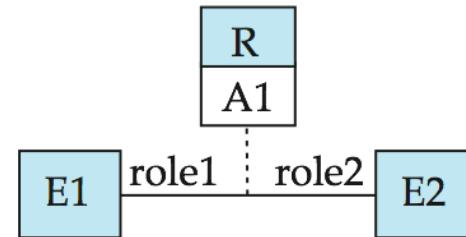
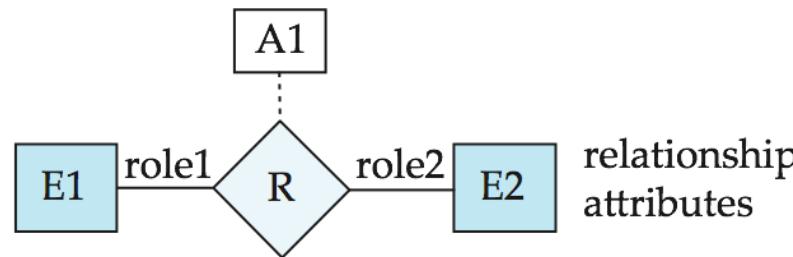


ER vs. UML Class Diagrams

ER Diagram Notation



Equivalent in UML

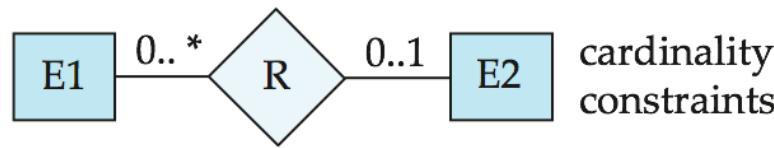


- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.
- The relationship set name may alternatively be written in a box, along with attributes of the relationship set, and the box is connected, using a dotted line, to the line depicting the relationship set.

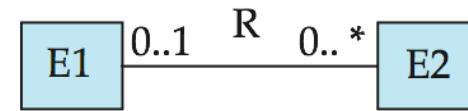


ER vs. UML Class Diagrams

ER Diagram Notation



Equivalent in UML

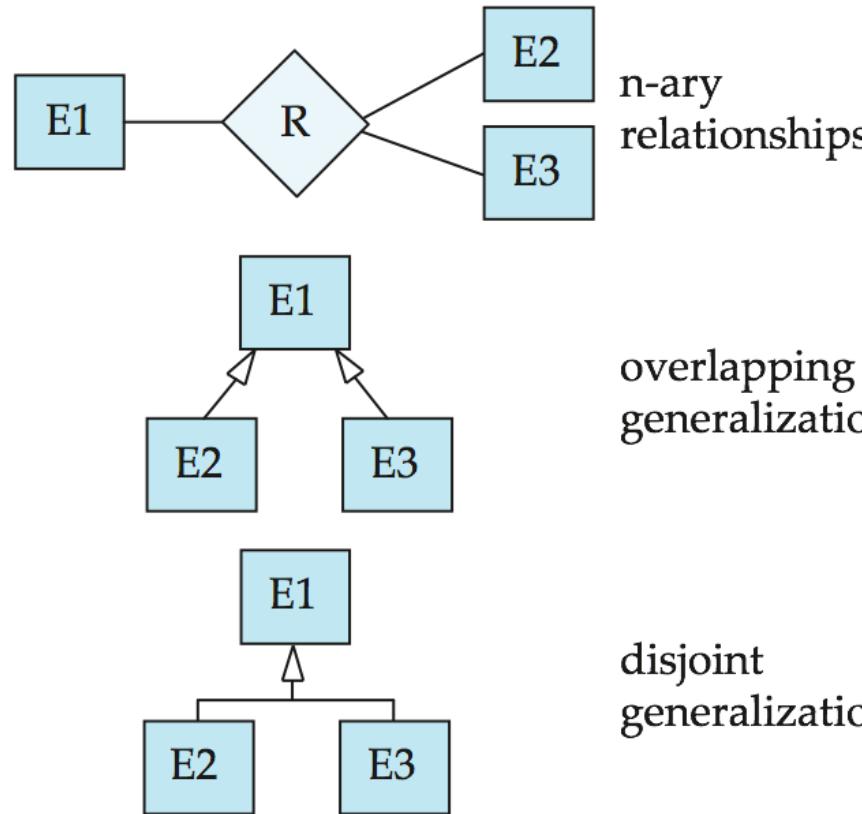


*Note reversal of position in cardinality constraint depiction

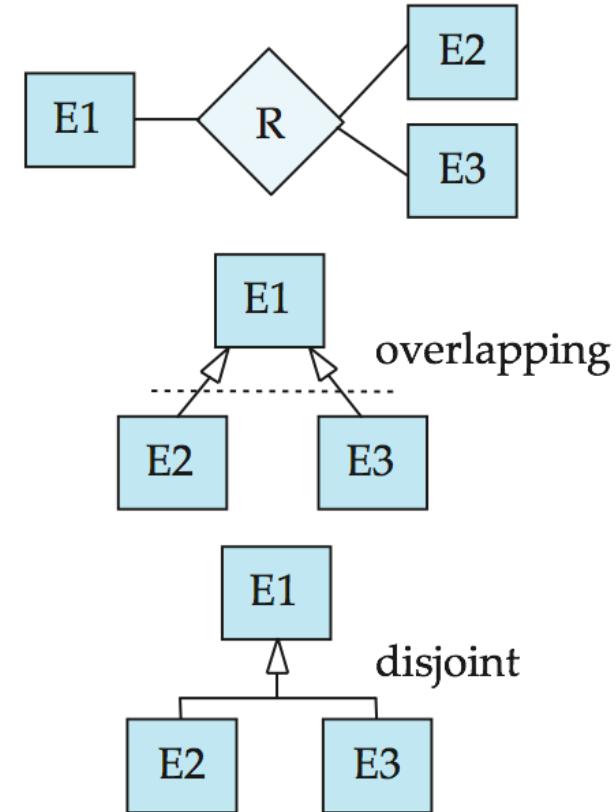


ER vs. UML Class Diagrams

ER Diagram Notation



Equivalent in UML



*Generalization can use merged or separate arrows independent of disjoint/overlapping



Questions?

Based on the slides of the course book



Database Systems

Lecture 8: Relational Database Design

Dr. Momtazi
momtazi@aut.ac.ir

Based on the slides of the course book



Outline

- **Features of Good Relational Design**
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
 - Functional Dependency Theory
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Good Relational Design

- In general, the goal of relational database design is to generate a set of relation schemas that allows us to **store information without unnecessary redundancy**, yet also allows us to **retrieve information easily**.
- This is accomplished by designing schemas that are in an appropriate *normal form*.



Overview of the University Schema

classroom(building, room_number, capacity)

department(dept_name, building, budget)

course(course_id, title, dept_name, credits)

instructor(ID, name, dept_name, salary)

section(course_id, sec_id, semester, year, building, room_number, time_slot_id)

teaches(ID, course_id, sec_id, semester, year)

student(ID, name, dept_name, tot_cred)

takes(ID, course_id, sec_id, semester, year, grade)

advisor(s_ID, i_ID)

time_slot(time_slot_id, day, start_time, end_time)

prereq(course_id, prereq_id)



Combine Schemas?

- Suppose we combine *instructor* and *department* into *inst_dept*
 - (*No connection to relationship set inst_dept*)
- Result is possible repetition of information

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000



Combine Schemas?

- Suppose we combine *instructor* and *department* into *inst_dept*
 - (*No connection to relationship set inst_dept*)
- Result has the following problems
 - Redundancy => risk of inconsistency
 - Incompleteness => problem of inserting new departments without any instructor



What About Smaller Schemas?

- Suppose we had started with *inst_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule, such as “each specific value for *dept_name* corresponds to at most one *building* and *budget*”. On other words, “if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key”
- This rule is specified as a **functional dependency**:
$$\textit{dept_name} \rightarrow \textit{building}, \textit{budget}$$

functional dependency reminds us to Super Key

- In *inst_dept*, the building and budget of a department may have to be repeated.
 - This indicates the need to decompose *inst_dept*

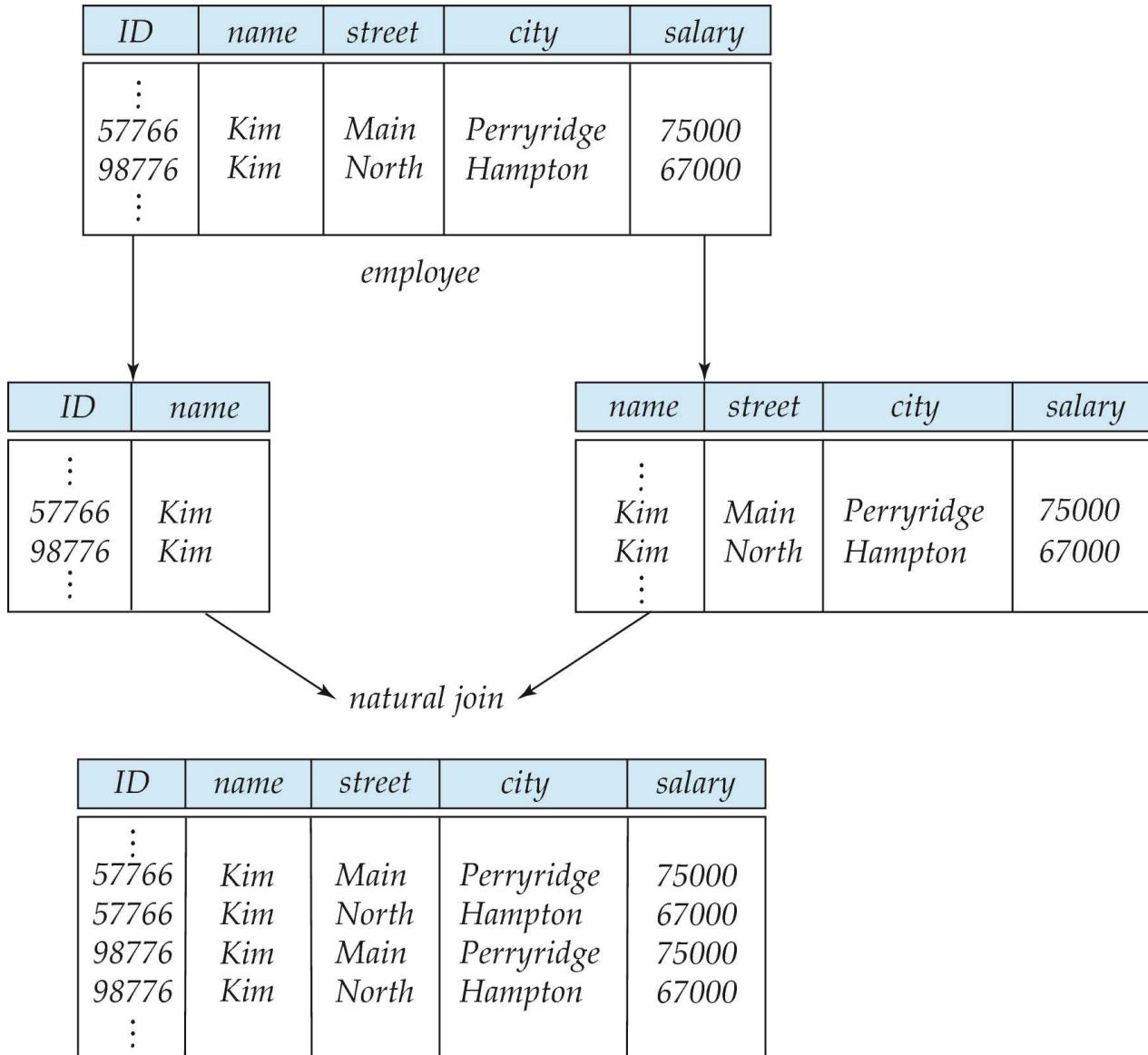


What About Smaller Schemas?

- Finding the right decomposition is hard for schemas with a large number of attributes and several functional dependencies.
- Not all decompositions are good.
- Suppose we decompose
 $\text{employee}(ID, name, street, city, salary)$ into
 $\text{employee1 } (ID, name)$
 $\text{employee2 } (name, street, city, salary)$
- Such decomposition results in losing information -- we cannot reconstruct the original employee relation -- and so, this is a **lossy decomposition**.



A Lossy Decomposition





Example of Lossless-Join Decomposition

- **Lossless join decomposition**

- Decomposition of $R = (A, B, C)$

$$R_1 = (A, B) \quad R_2 = (B, C)$$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B



Outline

- Features of Good Relational Design
- **Atomic Domains and First Normal Form**
- Decomposition Using Functional Dependencies
 - Functional Dependency Theory
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



First Normal Form

- Domain is **atomic** if its elements have no substring and are considered to be indivisible units
- Examples of non-atomic domains:
 - Set of names, composite attributes
 - Identification numbers like CS101 that can be broken up into parts
- CS --> computer science101 --> number of class
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic



First Normal Form

- Atomicity is actually a property of how the elements of the domain are used.
 - Example: Strings would normally be considered indivisible
 - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
 - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
 - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

NOTE: However, the database application still treats the domain as atomic, as long as it does not attempt to split the identifier and interpret parts of the identifier as a department abbreviation. The course schema stores the department name as a separate attribute, and the database application can use this attribute value to find the department of a course, instead of interpreting particular characters of the course identifier. Thus, our university schema can be considered to be in first normal form.



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- **Decomposition Using Functional Dependencies**
 - Functional Dependency Theory
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Goal — Devise a Theory for the Following

- Decide whether a particular relation R is in “good” form.
- In the case that a relation R is not in “good” form, decompose it into a set of relations $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation is in good form
 - the decomposition is a lossless-join decomposition
- Our theory is based on:
 - functional dependencies
 - multivalued dependencies



Functional Dependencies

- Defining all constraints (rules) on the data (the set of legal relations) in the real world.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a key.



Functional Dependencies

- Example: some of the constraints that are expected to hold in a university database
 - Students and instructors are uniquely identified by their ID.
 - Each student and instructor has only one name.
 - Each instructor and student is (primarily) associated with only one department.
 - Each department has only one value for its budget, and only one associated building.



Functional Dependencies

- Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α , they also agree on the attributes β . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A,B)$ with the following instance of r .

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.



Functional Dependencies

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
 - $K \rightarrow R$, and
 - for no $\alpha \subset K$, $\alpha \rightarrow R$ this implies that we are minimum
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys.



Functional Dependencies

- Consider the schema:

$inst_dept (ID, name, salary, dept_name, building, budget)$.

We expect these functional dependencies to hold:

$dept_name \rightarrow building$

$ID \rightarrow building$

but would not expect the following to hold:

$dept_name \rightarrow salary$

- We denote the fact that the pair of attributes ($ID, dept_name$) forms a superkey for $inst_dept$ by writing:

$ID, dept_name \rightarrow name, salary, building, budget$

super key



Use of Functional Dependencies

- We use functional dependencies to:
 - test relations to see if they are legal under a given set of functional dependencies.
 - ▶ If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F .
 - specify constraints on the set of legal relations
 - ▶ We say that F **holds on** R if all legal relations on R satisfy the set of functional dependencies F .



Use of Functional Dependencies

- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
- Example: In the following instance of the *classroom* relation we see the following functional dependency is satisfied.

$$\text{room_number} \rightarrow \text{capacity}$$

However, we believe that, in the real world, two classrooms in different buildings can have the same room number but with different room capacity.

<i>building</i>	<i>room_number</i>	<i>capacity</i>
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50



Use of Functional Dependencies

- Let us consider the following instance of relation r , to see which functional dependencies are satisfied.
 - Observe that $A \rightarrow C$ is satisfied.
 - ▶ There are two tuples that have an A value of $a1$. These tuples have the same C value—namely, $c1$.
 - ▶ Similarly, the two tuples with an A value of $a2$ have the same C value, $c2$.
 - ▶ There are no other pairs of distinct tuples that have the same A value.
 - The functional dependency $C \rightarrow A$ is not satisfied, however.
 - ▶ Consider the tuples $t4$ and $t5$.
 - ▶ These two tuples have the same C values, $c2$, but they have different A values, $a2$ and $a3$, respectively.
 - ▶ There is a pair of tuples $t4$ and $t5$ such that
$$t4[C] = t5[C], \text{ but } t4[A] \neq t5[A].$$

A	B	C	D
$a1$	b_1	c_1	d_1
$a1$	b_2	c_1	d_2
$a2$	b_2	c_2	d_2
$a2$	b_3	c_2	d_3
$a3$	b_3	c_2	d_4



Functional Dependencies

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
 - Example:
 - ▶ $ID, name \rightarrow ID$
 - ▶ $name \rightarrow name$
 - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$



Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For example: If $A \rightarrow B$ and $B \rightarrow C$,
then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^+ .
- F^+ is a superset of F .



Closure of a Set of Functional Dependencies

- We can find F^+ , the closure of F , by repeatedly applying **Armstrong's Axioms**:
 - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
 - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
 - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
 - **sound** (generate only functional dependencies that actually hold), and
 - **complete** (generate all functional dependencies that hold).



Example

- $R = (A, B, C, G, H, I)$

$$F = \{ A \rightarrow B$$

$$A \rightarrow C$$

$$CG \rightarrow H$$

$$CG \rightarrow I$$

$$B \rightarrow H\}$$

- some members of F^+

- $A \rightarrow H$

- ▶ by transitivity from $A \rightarrow B$ and $B \rightarrow H$

- $AG \rightarrow I$

- ▶ by augmenting $A \rightarrow C$ with G , to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$



Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F :

$$F^+ = F$$

repeat

for each functional dependency f in F^+

 apply reflexivity and augmentation rules on f

 add the resulting functional dependencies to F^+

for each pair of functional dependencies f_1 and f_2 in F^+

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

NOTE: We shall see an alternative procedure for this task later



Closure of Functional Dependencies

■ Additional rules:

- If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
- If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
- If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \delta$ holds, then $\alpha\beta \rightarrow \delta$ holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

$a \rightarrow c, b \rightarrow c \implies ab \rightarrow c$ correct
 $ab \rightarrow c \implies a \rightarrow c, b \rightarrow c$ incorrect



Example

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow B$
 $\quad A \rightarrow C$
 $\quad CG \rightarrow H$
 $\quad CG \rightarrow I$
 $\quad B \rightarrow H \}$

- some members of F^+
 - $AG \rightarrow I$
 - ▶ $A \rightarrow C \Rightarrow AG \rightarrow CG$
 - ▶ $CG \rightarrow I$
 - $CG \rightarrow HI$
 - ▶ union with $CG \rightarrow H$ and $CG \rightarrow I$



Closure of Attribute Sets

- Given a set of attributes α , define the ***closure*** of α **under** F (denoted by α^+) as the set of attributes that are functionally determined by α under F
- Algorithm to compute α^+ , the closure of α under F

```
result :=  $\alpha$ ;  
while (changes to result) do  
    for each  $\beta \rightarrow \gamma$  in  $F$  do  
        begin  
            if  $\beta \subseteq result$  then result := result  $\cup$   $\gamma$   
        end
```



Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$
 1. $result = AG$
 2. $result = ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)
 3. $result = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
 4. $result = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)



Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $(AG)^+ = ABCGHI$
- Is AG a candidate key?
 1. Is AG a superkey?
 1. Does $AG \rightarrow R? == \text{Is } (AG)^+ \supseteq R$ yes
 2. Is any subset of AG a superkey?
 1. Does $A \rightarrow R? == \text{Is } (A)^+ \supseteq R$ no
 2. Does $G \rightarrow R? == \text{Is } (G)^+ \supseteq R$ no
 - Usage of the attribute closure:
 - Testing for superkey:
 - ▶ To test if α is a superkey, we compute α^+ , and check if α^+ contains all attributes of R .

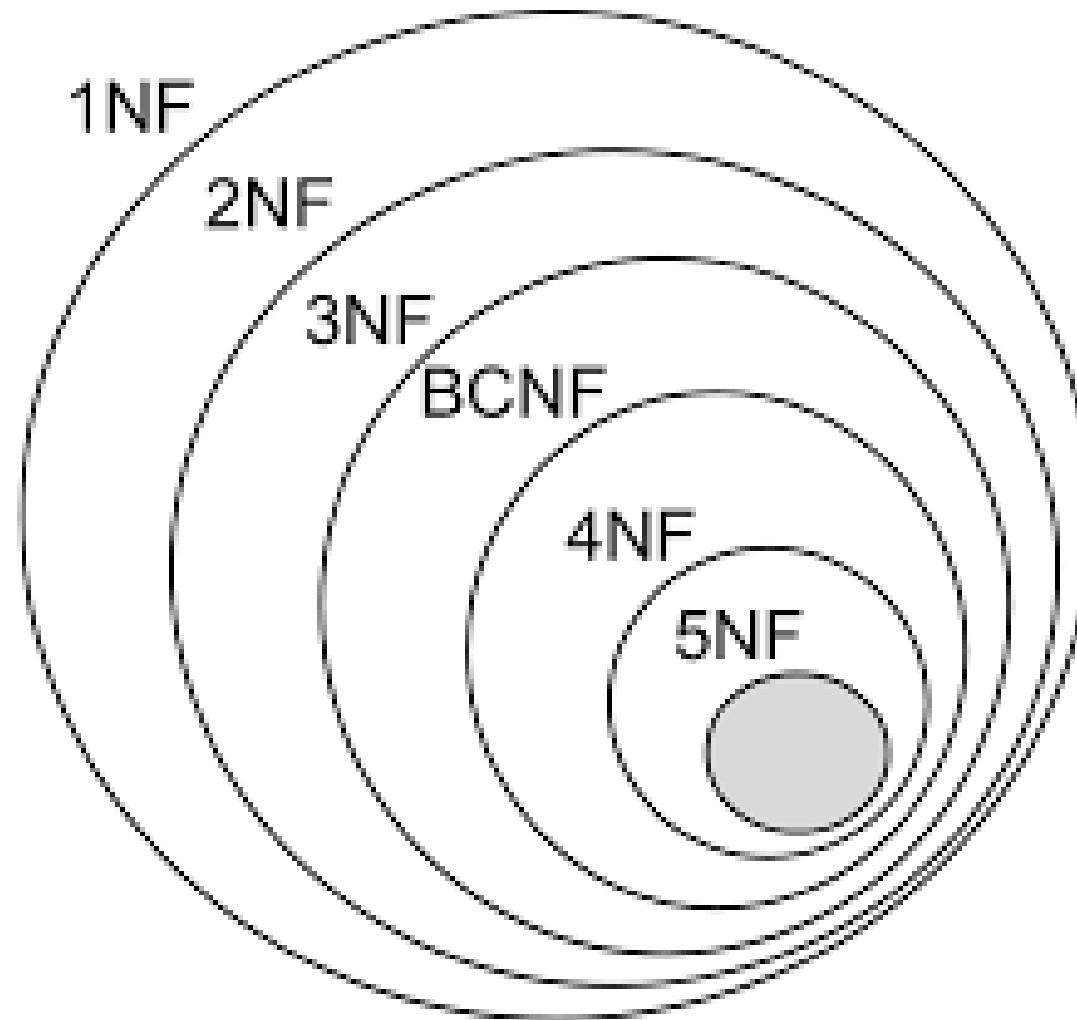


Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- **Decomposition Using Functional Dependencies**
 - Functional Dependency Theory
 - **Second Normal Form**
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Normal Forms





Second Normal Form (2NF)

- For a table to be in 2NF, there are two requirements
 - The database is in first normal form
 - All nonkey attributes in the table must be functionally dependent on the entire primary key

Note: Remember that we are dealing with non-key attributes



Second Normal Form (2NF)

Example 1

$R = \{\text{Title}, \text{PubId}, \text{Auld}, \text{Price}, \text{AuAddress}\}$

Key: $\{\text{Title}, \text{PubId}, \text{Auld}\}$

1. $\{\text{Title}, \text{PubId}, \text{AulD}\} \rightarrow \{\text{Price}\}$
2. $\{\text{AulD}\} \rightarrow \{\text{AuAddress}\}$

- AuAddress does not belong to a key
- AuAddress functionally depends on Auld which is a subset of a key



2NF - Decomposition

1. If a data item is fully functionally dependent on only a part of the primary key, move that data item and that part of the primary key to a new table.
2. If other data items are functionally dependent on the same part of the key, place them in the new table also
3. Make the partial primary key copied from the original table the primary key for the new table. Place all items that appear in the repeating group in a new table



2NF - Decomposition

Example 1

$R = \{\text{Title}, \text{PubId}, \text{Auld}, \text{Price}, \text{AuAddress}\}$

Key: $\{\text{Title}, \text{PubId}, \text{Auld}\}$

1. $\{\text{Title}, \text{PubId}, \text{AulD}\} \rightarrow \{\text{Price}\}$
2. $\{\text{AulD}\} \rightarrow \{\text{AuAddress}\}$

- AuAddress does not belong to a key
- AuAddress functionally depends on Auld which is a subset of a key

Decomposition

Old Scheme = {Title, PubId, AuId, Price, AuAddress}

New Scheme = {Title, PubId, AuId, Price}

New Scheme = {AuId, AuAddress}



2NF - Decomposition

Example 2

R = {City, Street, HouseNumber, HouseColor, CityPopulation, CityCode}

Key: {City, Street, HouseNumber}

1. $\{City, Street, HouseNumber\} \rightarrow \{HouseColor\}$
2. $\{City\} \rightarrow \{CityPopulation\}$
3. $\{City\} \rightarrow \{CityCode\}$

- CityPopulation does not belong to any key.
- CityPopulation is functionally dependent on the City which is a proper subset of the key

Decomposition

Old Scheme = {City, Street, HouseNumber, HouseColor, CityPopulation, CityCode}

New Scheme = {City, Street, HouseNumber, HouseColor}

New Scheme = {City, CityPopulation, CityCode}



2NF - Decomposition

Example 3

$R = \{\text{studio, movie, budget, studio_city}\}$

Key: $\{\text{studio, movie}\}$

1. $\{\text{studio, movie}\} \rightarrow \{\text{budget}\}$
2. $\{\text{studio}\} \rightarrow \{\text{studio_city}\}$

- studio_city is not a part of a key
- studio_city functionally depends on studio which is a proper subset of the key

Decomposition

Old Scheme = Studio, Movie, Budget, StudioCity}

New Scheme = Movie, Studio, Budget}

New Scheme = Studio, City}



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- **Decomposition Using Functional Dependencies**
 - Functional Dependency Theory
 - Second Normal Form
 - **Boyce-Codd Normal Form**
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R



Boyce-Codd Normal Form

n Example

- | The *instr_dep* schema is **not** in BCNF:

instr_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

- ▶ because $\text{dept_name} \rightarrow \text{building}, \text{budget}$ holds on *instr_dept*, but *dept_name* is not a superkey

- | The *instructor* schema is **in** BCNF.

instructor (*ID*, *name*, *salary*, *dept_name*)

- ▶ Because all of the nontrivial functional dependencies that hold, such as $\text{ID} \rightarrow \text{name}$, *dept_name*, *salary* include *ID* on the left side of the arrow, and *ID* is a superkey for *instructor*.



Decomposing a Schema into BCNF

- n Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose R into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

- n In our example,

- | $\alpha = \text{dept_name}$
- | $\beta = \text{building, budget}$

and inst_dept is replaced by

- | $(\alpha \cup \beta) = (\text{dept_name, building, budget})$
- | $(R - (\beta - \alpha)) = (\text{ID, name, salary, dept_name})$

NOTE: When we decompose a schema that is not in BCNF, it may be that one or more of the resulting schemas are not in BCNF. In such cases, further decomposition is required, the eventual result of which is a set of BCNF schemas.



BCNF Example 1

n Example:

- | Each student can contribute in various running projects in a university
- | Each student has a unique email address for all his/hir correspondences in the project

⇒ *std_project* relation

std_project(s_ID, p_ID, s_email)

key?



BCNF Example 1

n Example:

- | Each student can contribute in various running projects in a university
- | Each student has a unique email address for all his/hir correspondences in the project

⇒ *std_project* relation

std_project(s_ID, p_ID, s_email)

key?

s_ID, p_ID



BCNF Example 1

n Example:

- | Each student can contribute in various running projects in a university
- | Each student has a unique email address for all his/hir correspondences in the project

⇒ *std_project* relation

std_project(s_ID, p_ID, s_email)

functional dependencies?



BCNF Example 1

n Example:

- | Each student can contribute in various running projects in a university
- | Each student has a unique email address for all his/hir correspondences in the project

⇒ *std_project* relation

std_project(s_ID, p_ID, s_email)

functional dependencies?

$s_ID \rightarrow s_email$



BCNF Example 1

n Example:

$std_project(s_ID, p_ID, s_email)$

$s_ID \rightarrow s_email$

$std_project$ is not in BCNF because s_ID is not a superkey

=> BCNF decomposition

(s_ID, s_email)

$(s_ID, proj_ID)$

Both the above schemas are BCNF.



BCNF Example 2

- Example:

- An instructor can be associated with only a single department
- A student may have more than one advisor, but at most one from a given department.

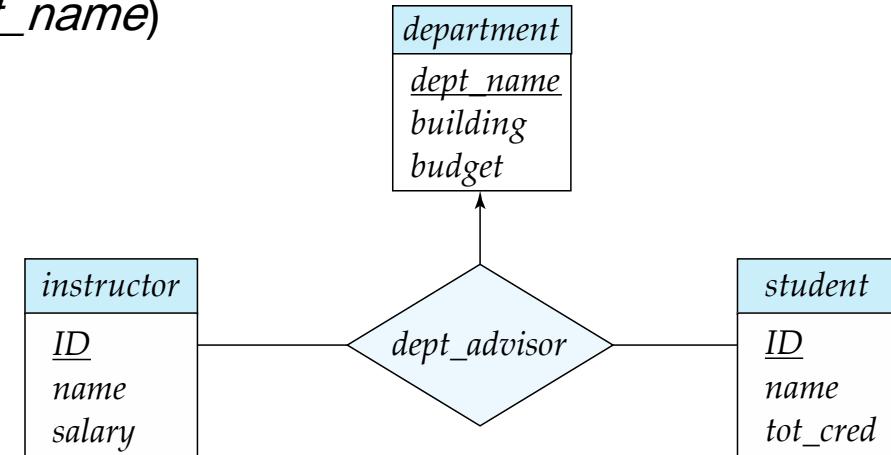
(Such an arrangement makes sense for students with a double major.)

⇒ *dep_advise* relation:

many-to-one from the pair $\{student, instructor\}$ to *department*

dep_advisor (*s_ID*, *i_ID*, *dept_name*)

key?





BCNF Example 2

- Example:

- An instructor can be associated with only a single department
- A student may have more than one advisor, but at most one from a given department.

(Such an arrangement makes sense for students with a double major.)

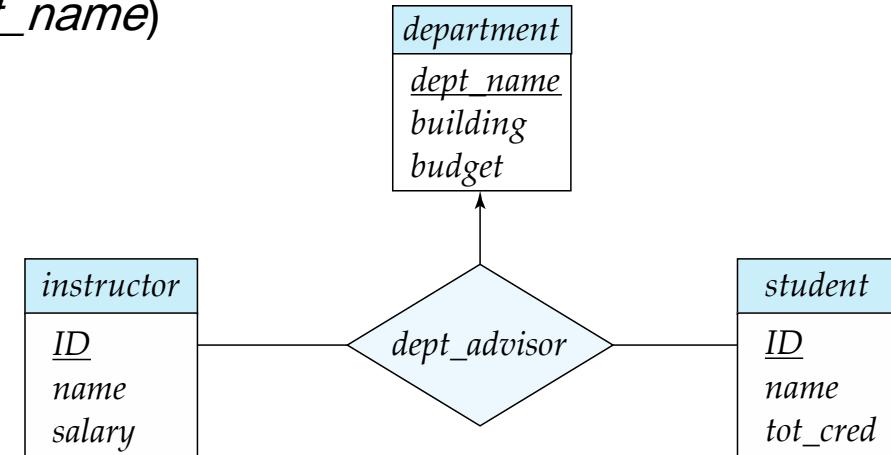
⇒ *dep_advise* relation:

many-to-one from the pair {*student*, *instructor*} to *department*

dep_advisor (*s_ID*, *i_ID*, *dept_name*)

key?

s_ID, *dept_name*





BCNF Example 2

- Example:

- An instructor can be associated with only a single department
- A student may have more than one advisor, but at most one from a given department.

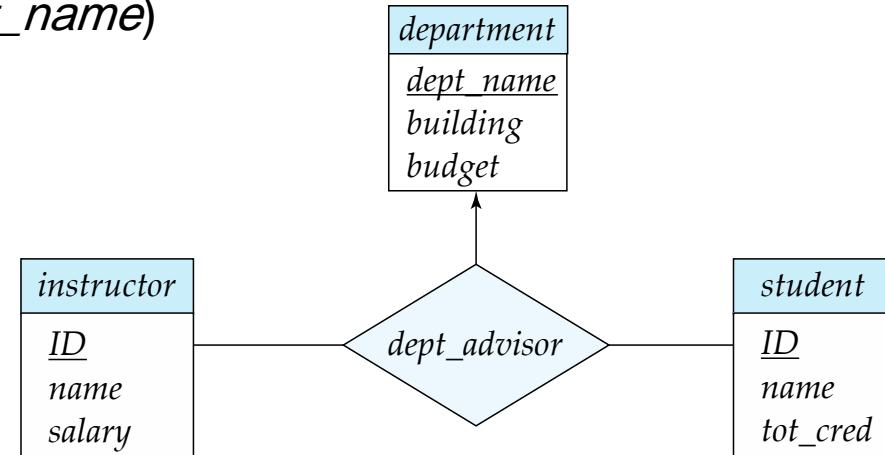
(Such an arrangement makes sense for students with a double major.)

⇒ *dep_advise* relation:

many-to-one from the pair {*student*, *instructor*} to *department*

dep_advisor (*s_ID*, *i_ID*, *dept_name*)

functional dependencies?





BCNF Example 2

- Example:

- An instructor can be associated with only a single department
- A student may have more than one advisor, but at most one from a given department.

(Such an arrangement makes sense for students with a double major.)

⇒ *dep_advise* relation:

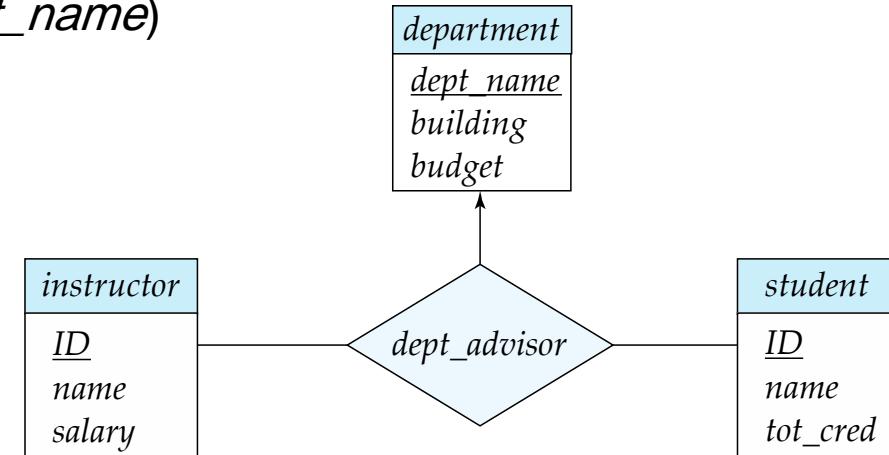
many-to-one from the pair {*student*, *instructor*} to *department*

dep_advisor (*s_ID*, *i_ID*, *dept_name*)

functional dependencies?

$i_ID \rightarrow dept_name$

$s_ID, dept_name \rightarrow i_ID$





BCNF Example 2

- Example:

$dep_advisor(s_ID, i_ID, dept_name)$

$i_ID \rightarrow dept_name$

$s_ID, dept_name \rightarrow i_ID$

$dep_advisor$ is not in BCNF because i_ID is not a superkey

=> BCNF decomposition



BCNF Example 2

- Example:

$dep_advisor(s_ID, i_ID, dept_name)$

$i_ID \rightarrow dept_name$

$s_ID, dept_name \rightarrow i_ID$

this composition is not dependency preserve.

$dep_advisor$ is not in BCNF because i_ID is not a superkey

=> BCNF decomposition

(s_ID, i_ID)

$(i_ID, dept_name)$

Both the above schemas are BCNF. However, there is no schema that includes all the attributes appearing in the functional dependency

$s_ID, dept_name \rightarrow i_ID$



BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a **weaker** normal form, known as *third normal form*.



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- **Decomposition Using Functional Dependencies**
 - Functional Dependency Theory
 - Second Normal Form
 - Boyce-Codd Normal Form
 - **Third Normal Form**
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .

(**NOTE**: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF
 - because in BCNF one of the first two conditions above must hold.
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).



Third Normal Form

n Example:

$dep_advisor(s_ID, i_ID, dept_name)$

$i_ID \rightarrow dept_name$

$s_ID, dept_name \rightarrow i_ID$

$dep_advisor$ is in 3NF because dep_name is contained in a candidate key
=> no decomposition is needed

n Conclusion:

We have seen the trade-off that must be made between BCNF and 3NF when there is no dependency-preserving BCNF design.

BCNF --> minimal space
3NF --> minimal time coplex



Example

- $R = (A, B, C, D)$
- $F = \{AB \rightarrow C$
 $D \rightarrow B$
 $AC \rightarrow D\}$
- See the solutions on the board
 - Candidate keys
 - Is $AD \rightarrow B$ included in F^+
 - *Is R in BCNF?*
 - *Is R in 3NF?*
 - *Decompose R into BCNF*



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- **Decomposition Using Functional Dependencies**
 - Functional Dependency Theory
 - Second Normal Form
 - Boyce-Codd Normal Form
 - Third Normal Form
 - **Normalization's Goal**
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in “good” form.
- In the case that a relation scheme R is not in “good” form, decompose it into a set of relation scheme $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation scheme is in good form
 - the decomposition is a lossless-join decomposition
 - Preferably, the decomposition should be dependency preserving.



Functional-Dependency Theory

- Considering the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- Developing algorithms to generate lossless decompositions into BCNF and 3NF
- Developing algorithms to test if a decomposition is dependency-preserving



Canonical Cover

- By any update performed by users on a relation schema, the database system must ensure that the update does not violate any functional dependencies; i.e., all the functional dependencies should be satisfied in the new database state.
- The system must roll back the update if it violates any functional dependencies.
- We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set.
- Any database that satisfies the simplified set of functional dependencies also satisfies the original set, and vice versa, since the two sets have the same closure. However, the simplified set is easier to test.



Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
 - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
 - Parts of a functional dependency may be redundant
 - ▶ E.g.: on RHS: $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
 - ▶ E.g.: on LHS: $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F , having no redundant dependencies or redundant parts of dependencies



Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
 - Attribute A is **extraneous** in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
 - Attribute A is **extraneous** in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F .
- *Note:* implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one



Extraneous Attributes

- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
 - B is extraneous in $AB \rightarrow C$
 - ▶ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (i.e. the result of dropping B from $AB \rightarrow C$).

- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
 - C is extraneous in $AB \rightarrow CD$
 - ▶ because $AB \rightarrow C$ can be inferred even after deleting C



Testing if an Attribute is Extraneous

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F
- To test if attribute $A \in \alpha$ is extraneous in α
 1. compute $(\{\alpha\} - A)^+$ using the dependencies in F
 2. check that $(\{\alpha\} - A)^+$ contains β ; if it does, A is extraneous in α
- To test if attribute $A \in \beta$ is extraneous in β
 1. compute α^+ using only the dependencies in
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
 2. check that α^+ contains A ; if it does, A is extraneous in β



Extraneous Attributes

■ Example:

- Given $F = \{A \rightarrow C, AB \rightarrow C\}$
- B is extraneous in $AB \rightarrow C$
 - ▶ $(\{\alpha\} - A)^+$ under F contains β
 - ▶ $A^+ = AC$



Extraneous Attributes

■ Example:

- Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
- C is extraneous in $AB \rightarrow CD$
 - α^+ under F' contains C
 - $(AB)^+ = ABCD$



Extraneous Attributes

■ Example:

- Given $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$
- C is extraneous in $AB \rightarrow CD$
 - α^+ under F' contains C
 - $(AB)^+ = ABCDE$



Canonical Cover

- A **canonical cover** for F is a set of dependencies F_c such that
 - F logically implies all dependencies in F_c , and
 - F_c logically implies all dependencies in F , and
 - No functional dependency in F_c contains an extraneous attribute, and
 - Each left side of functional dependency in F_c is unique.



Canonical Cover

- To compute a canonical cover for F :

repeat

 Use the union rule to replace any dependencies in F

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$

 Find a functional dependency $\alpha \rightarrow \beta$ with an

 extraneous attribute either in α or in β

 /* Note: test for extraneous attributes done using F_c , not F^* */

 If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

until F does not change

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied
- Note: Whenever the right hand side become empty the entire functional dependency should be removed



Computing a Canonical Cover

- $R = (A, B, C)$
 $F = \{A \rightarrow BC$
 $\quad B \rightarrow C$
 $\quad A \rightarrow B$
 $\quad AB \rightarrow C\}$



Computing a Canonical Cover

- $R = (A, B, C)$
 $F = \{A \rightarrow BC$
 $\quad B \rightarrow C$
 $\quad A \rightarrow B$
 $\quad AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
 - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
 - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
 - ▶ Yes: in fact, $B \rightarrow C$ is already present!
 - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
 - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
 - ▶ Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
 - Can use attribute closure of A in more complex cases
- The canonical cover is:
 - $A \rightarrow B$
 - $B \rightarrow C$



Computing a Canonical Cover

- $R = (A, B, C)$
 $F = \{A \rightarrow BC$
 $B \rightarrow AC$
 $C \rightarrow AB\}$

- Computations on the board



Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+ :
 - $R_1 \cap R_2 \rightarrow R_1$
 - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies



Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B$
 $B \rightarrow C\}$
- Key = {A}
- R is not in BCNF
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
 - R_1 and R_2 in BCNF
 - Lossless-join decomposition
$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$
 - Dependency preserving

BCNF and 3NF never produce lossless-join



Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i ,
 - A decomposition is **dependency preserving**, if
$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$
 - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.



Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into R_1, R_2, \dots, R_n we apply the following test (with attribute closure done with respect to F)
 - $result = \alpha$
while (changes to $result$) do
 for each R_i in the decomposition
 $t = (result \cap R_i)^+ \cap R_i$
 $result = result \cup t$
 - If $result$ contains all attributes in β , then the functional dependency $\alpha \rightarrow \beta$ is preserved.
- We apply the test on all dependencies in F to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \dots \cup F_n)^+$



Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
 1. compute α^+ (the attribute closure of α), and
 2. verify that it includes all attributes of R , that is, it is a superkey of R .
- **Simplified test:** To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+ .
 - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.



Testing for BCNF

- Simplified test using only F is incorrect when testing a relation in a decomposition of R
- Example:
 - $R = (A, B, C, D, E)$,
 - $F = \{ A \rightarrow B, BC \rightarrow D \}$

Decompose R into

- ▶ $R_1 = (A, B)$
- ▶ $R_2 = (A, C, D, E)$



Testing for BCNF

- Simplified test using only F is incorrect when testing a relation in a decomposition of R
- Example:
 - $R = (A, B, C, D, E)$,
 - $F = \{ A \rightarrow B, BC \rightarrow D \}$

Decompose R into

- ▶ $R_1 = (A, B)$
- ▶ $R_2 = (A, C, D, E)$

- ▶ Neither of the dependencies in F contain only attributes from (A, C, D, E)
 - so we might be misled into thinking R_2 satisfies BCNF.
 - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.



Testing Decomposition for BCNF

- To check if a relation R_i in a decomposition of R is in BCNF,
 - Either test R_i for BCNF with respect to the **restriction** of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
 - or use the original set of dependencies F that hold on R , but with the following test:
 - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i
 - ▶ If the condition is violated by some $\alpha \rightarrow \beta$ in F , the dependency
$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$
can be shown to hold on R_i , and R_i violates BCNF.
 - ▶ We use above dependency to decompose R_i



BCNF Decomposition Algorithm

```
result := {R};  
done := false;  
compute  $F^+$ ;  
while (not done) do  
  if (there is a schema  $R_i$  in result that is not in BCNF)  
    then begin  
      let  $\alpha \rightarrow \beta$  be a nontrivial functional dependency that  
      holds on  $R_i$  such that  $\alpha \rightarrow R_i$  is not in  $F^+$ ,  
      and  $\alpha \cap \beta = \emptyset$ ;  
      result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );  
    end  
  else done := true;
```

Note: each R_i is in BCNF, and decomposition is lossless-join.



Example of BCNF Decomposition

- *class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)*
- Functional dependencies:
 - ?
- A candidate key:
 - ?



Example of BCNF Decomposition

- $\text{class}(\text{course_id}, \text{title}, \text{dept_name}, \text{credits}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{capacity}, \text{time_slot_id})$
- Functional dependencies:
 - $\text{course_id} \rightarrow \text{title}, \text{dept_name}, \text{credits}$
 - $\text{building}, \text{room_number} \rightarrow \text{capacity}$
 - $\text{course_id}, \text{sec_id}, \text{semester}, \text{year} \rightarrow \text{building}, \text{room_number}, \text{time_slot_id}$
- A candidate key:
 - ?



Example of BCNF Decomposition

- $\text{class}(\text{course_id}, \text{title}, \text{dept_name}, \text{credits}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{capacity}, \text{time_slot_id})$
- Functional dependencies:
 - $\text{course_id} \rightarrow \text{title}, \text{dept_name}, \text{credits}$
 - $\text{building}, \text{room_number} \rightarrow \text{capacity}$
 - $\text{course_id}, \text{sec_id}, \text{semester}, \text{year} \rightarrow \text{building}, \text{room_number}, \text{time_slot_id}$
- A candidate key:
 - $\{\text{course_id}, \text{sec_id}, \text{semester}, \text{year}\}$.



Example of BCNF Decomposition

- *class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)*
- BCNF Decomposition:
 - ?



Example of BCNF Decomposition

- $\text{class}(\text{course_id}, \text{title}, \text{dept_name}, \text{credits}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{capacity}, \text{time_slot_id})$
- BCNF Decomposition:
 - $\text{course_id} \rightarrow \text{title}, \text{dept_name}, \text{credits}$
 - ▶ course_id is not a superkey.
 - We replace class by:
 - ▶ $\text{course}(\text{course_id}, \text{title}, \text{dept_name}, \text{credits})$
 - ▶ $\text{class-1}(\text{course_id}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{capacity}, \text{time_slot_id})$



BCNF Decomposition

- *course* is in BCNF
- BCNF Decomposition for class-1:
 - ?



BCNF Decomposition

- *course* is in BCNF
- BCNF Decomposition for *class-1*:
 - $\text{building}, \text{room_number} \rightarrow \text{capacity}$
 - ▶ $\{\text{building}, \text{room_number}\}$ is not a superkey for *class-1*.
 - We replace *class-1* by:
 - ▶ *classroom* ($\text{building}, \text{room_number}, \text{capacity}$)
 - ▶ *section* ($\text{course_id}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{time_slot_id}$)
- *classroom* and *section* are in BCNF.



Third Normal Form: Motivation

- There are some situations where
 - BCNF is not dependency preserving, and
 - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
 - Allows some redundancy (with resultant problems; we will see examples later)
 - But functional dependencies can be checked on individual relations without computing a join.
 - There is always a lossless-join, dependency-preserving decomposition into 3NF.



3NF Example

■ Relation *dept_advisor*.

- $\text{dept_advisor}(s_ID, i_ID, \text{dept_name})$
- $F = \{s_ID, \text{dept_name} \rightarrow i_ID,$
 $i_ID \rightarrow \text{dept_name}\}$
- Two candidate keys: s_ID , dept_name , and i_ID , s_ID
- R is in 3NF
 - ▶ $s_ID, \text{dept_name} \rightarrow i_ID$
 - $s_ID, \text{dept_name}$ is a superkey
 - ▶ $i_ID \rightarrow \text{dept_name}$
 - dept_name is contained in a candidate key



Redundancy in 3NF

- There is some redundancy in this schema

s_ID	i_ID	dept_name
A	X	M
B	X	M
C	X	M
Null	Y	N

- repetition of information (e.g., the relationship X, M)
 - $(i_ID, dept_name)$
- need to use null values (e.g., to represent the relationship Y, N where there is no corresponding value for s_ID).



Testing for 3NF

- Optimization: Need to check only FDs in F , need not check all FDs in F^+ .
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
 - this test is rather more expensive, since it involve finding candidate keys
 - testing for 3NF has been shown to be NP-hard
 - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time



3NF Decomposition Algorithm

Let F_c be a canonical cover for F ;

$i := 0$;

for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**

if none of the schemas R_j , $1 \leq j \leq i$ contains $\alpha \beta$

then begin

$i := i + 1$;

$R_i := \alpha \beta$

end

if none of the schemas R_j , $1 \leq j \leq i$ contains a candidate key for R

then begin

$i := i + 1$;

$R_i :=$ any candidate key for R ;

end

/* Optionally, remove redundant relations */

repeat

if any schema R_j is contained in another schema R_k

then /* delete R_j */

$R_j = R_{j+1}$;

$i = i - 1$;

return (R_1, R_2, \dots, R_i)



3NF Decomposition Algorithm

- n Above algorithm ensures:
 - | each relation schema R_i is in 3NF
 - | decomposition is dependency preserving and lossless-join



Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
 - the decomposition is lossless
 - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
 - the decomposition is lossless
 - it may not be possible to preserve dependencies.



Design Goals

- Goal for a relational database design is:
 - BCNF.
 - Lossless join.
 - Dependency preservation.
- If we cannot achieve this, we accept one of
 - Lack of dependency preservation
 - Redundancy due to use of 3NF



Design Goals

- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.
- Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
 - Functional Dependency Theory
 - Second Normal Form
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- **Decomposition Using Multivalued Dependencies**
- More Normal Form
- Database-Design Process
- Modeling Temporal Data



Multivalued Dependencies

- Suppose we record names of children, and phone numbers for instructors:
 - $inst_child(ID, child_name)$
 - $inst_phone(ID, phone_number)$
- If we were to combine these schemas to get
 - $inst_info(ID, child_name, phone_number)$
 - Example data:
 - (99999, William, 512-555-1234)
 - (99999, David, 512-555-4321)
 - (99999, David, 512-555-1234)
 - (99999, William, 512-555-4321)
- This relation is in BCNF
 - Why?



How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

inst_info (ID, child_name, phone)

- where an instructor may have more than one phone and can have multiple children

<i>ID</i>	<i>child_name</i>	<i>phone</i>
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	Willian	512-555-4321

inst_info



How good is BCNF?

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

(99999, David, 981-992-3443)
(99999, William, 981-992-3443)



How good is BCNF?

- Therefore, it is better to decompose *inst_info* into:

	<i>ID</i>	<i>child_name</i>
<i>inst_child</i>	99999	David
	99999	David
	99999	William
	99999	Willian

	<i>ID</i>	<i>phone</i>
<i>inst_phone</i>	99999	512-555-1234
	99999	512-555-4321
	99999	512-555-1234
	99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later.



Multivalued Dependencies (MVDs)

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_3[R - \beta] = t_2[R - \beta]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_4[R - \beta] = t_1[R - \beta]$$



MVD (Cont.)

- Tabular representation of $\alpha \rightarrow\!\!\!\rightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$



Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

Y, Z, W

- We say that $Y \twoheadrightarrow Z$ (Y **multidetermines** Z) if and only if for all possible relations $r(R)$

$$\langle y_1, z_1, w_1 \rangle \in r \text{ and } \langle y_1, z_2, w_2 \rangle \in r$$

then

$$\langle y_1, z_1, w_2 \rangle \in r \text{ and } \langle y_1, z_2, w_1 \rangle \in r$$

- Note that since the behavior of Z and W are identical it follows that $Y \twoheadrightarrow Z$ if $Y \twoheadrightarrow W$



Example (Cont.)

- In our example:

$ID \rightarrow\rightarrow child_name$

$ID \rightarrow\rightarrow phone_number$

- The above formal definition is supposed to formalize the notion that given a particular value of $Y (ID)$ it has associated with it a set of values of Z ($child_name$) and a set of values of W ($phone_number$), and these two sets are in some sense independent of each other.
- Note:
 - If $Y \rightarrow Z$ then $Y \rightarrow\rightarrow Z$
 - Indeed we have (in above notation) $Z_1 = Z_2$
The claim follows.



Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
 1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
 2. To specify **constraints** on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation r fails to satisfy a given multivalued dependency, we can construct a relations r' that does satisfy the multivalued dependency by adding tuples to r .



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
 - Functional Dependency Theory
 - Second Normal Form
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- **More Normal Form**
- Database-Design Process
- Modeling Temporal Data



Fourth Normal Form

- A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
 - $\alpha \twoheadrightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
 - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF



Restriction of Multivalued Dependencies

- The restriction of D to R_i is the set D_i consisting of
 - All functional dependencies in D^+ that include only attributes of R_i
 - All multivalued dependencies of the form
$$\alpha \rightarrow\!\!\!\rightarrow (\beta \cap R_i)$$
where $\alpha \subseteq R_i$ and $\alpha \rightarrow\!\!\!\rightarrow \beta$ is in D^+



4NF Decomposition Algorithm

result := { R };

done := false;

compute D^+ ;

Let D_i denote the restriction of D^+ to R_i

while (*not done*)

if (there is a schema R_i in *result* that is not in 4NF) **then**

begin

let $\alpha \rightarrow\!\!\rightarrow \beta$ be a nontrivial multivalued dependency that holds
on R_i such that $\alpha \rightarrow R_i$ is not in D_i , and $\alpha \cap \beta = \emptyset$;

result := (*result* - R_i) \cup (R_i - β) \cup (α, β);

end

else *done* := true;

Note: each R_i is in 4NF, and decomposition is lossless-join





Example

- $R = (A, B, C, G, H, I)$

$$F = \{ A \rightarrow\!\!\! \rightarrow B$$

$$B \rightarrow\!\!\! \rightarrow HI$$

$$CG \rightarrow\!\!\! \rightarrow H \}$$



Example

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow\!\!\!\rightarrow B$
 $\quad B \rightarrow\!\!\!\rightarrow HI$
 $\quad CG \rightarrow\!\!\!\rightarrow H \}$
- R is not in 4NF since $A \rightarrow\!\!\!\rightarrow B$ and A is not a superkey for R
- Decomposition
 - a) $R_1 = (A, B)$ $(R_1$ is in 4NF)
 - b) $R_2 = (A, C, G, H, I)$ $(R_2$ is not in 4NF => decompose it)



Example

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow\!\!\!\rightarrow B$
 $\quad B \rightarrow\!\!\!\rightarrow HI$
 $\quad CG \rightarrow\!\!\!\rightarrow H \}$
- R is not in 4NF since $A \rightarrow\!\!\!\rightarrow B$ and A is not a superkey for R
- Decomposition
 - a) $R_1 = (A, B)$ $(R_1$ is in 4NF)
 - b) $R_2 = (A, C, G, H, I)$ $(R_2$ is not in 4NF => decompose it)
 - c) $R_{21} = (C, G, H)$ $(R_{21}$ is in 4NF)
 - d) $R_{22} = (A, C, G, I)$ $(R_{22}$ is not in 4NF => decompose it)



Example

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow\!\!\!\rightarrow B$
 $\quad B \rightarrow\!\!\!\rightarrow HI$
 $\quad CG \rightarrow\!\!\!\rightarrow H \}$
- R is not in 4NF since $A \rightarrow\!\!\!\rightarrow B$ and A is not a superkey for R
- Decomposition
 - a) $R_1 = (A, B)$ $(R_1$ is in 4NF)
 - b) $R_2 = (A, C, G, H, I)$ $(R_2$ is not in 4NF \Rightarrow decompose it)
 - c) $R_{21} = (C, G, H)$ $(R_{21}$ is in 4NF)
 - d) $R_{22} = (A, C, G, I)$ $(R_{22}$ is not in 4NF \Rightarrow decompose it)
 - $A \rightarrow\!\!\!\rightarrow B$ and $B \rightarrow\!\!\!\rightarrow HI \Rightarrow A \rightarrow\!\!\!\rightarrow HI$, (MVD transitivity)
 - hence $A \rightarrow\!\!\!\rightarrow I$ (*MVD restriction to R_{22}*)



Example

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow\!\!\!\rightarrow B$
 $\quad B \rightarrow\!\!\!\rightarrow HI$
 $\quad CG \rightarrow\!\!\!\rightarrow H \}$
- R is not in 4NF since $A \rightarrow\!\!\!\rightarrow B$ and A is not a superkey for R
- Decomposition
 - a) $R_1 = (A, B)$ $(R_1$ is in 4NF)
 - b) $R_2 = (A, C, G, H, I)$ $(R_2$ is not in 4NF => decompose it)
 - c) $R_{21} = (C, G, H)$ $(R_{21}$ is in 4NF)
 - d) $R_{22} = (A, C, G, I)$ $(R_{22}$ is not in 4NF => decompose it)
 - $A \rightarrow\!\!\!\rightarrow B$ and $B \rightarrow\!\!\!\rightarrow HI \Rightarrow A \rightarrow\!\!\!\rightarrow HI$, (MVD transitivity)
 - hence $A \rightarrow\!\!\!\rightarrow I$ (*MVD restriction to R_{22}*)
 - e) $R_{221} = (A, I)$ $(R_{221}$ is in 4NF)
 - f) $R_{222} = (A, C, G)$ $(R_{222}$ is in 4NF)



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
 - Functional Dependency Theory
 - Second Normal Form
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- **Database-Design Process**
- Modeling Temporal Data



Overall Database Design Process

- We have assumed schema R is given
 - R could have been generated when converting E-R diagram to a set of tables.
 - R could have been a single relation containing *all* attributes that are of interest (called **universal relation**). Normalization breaks R into smaller relations.
 - R could have been the result of some ad hoc design of relations, which we then test/convert to normal form.



ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
 - Example: an *employee* entity with attributes *department_name* and *building*, and a functional dependency $\text{department_name} \rightarrow \text{building}$
 - Good design would have made department an entity



Naming of Attributes

- Each attribute name should have a unique meaning in the database.
- This prevents us from using the same attribute to mean different things in different schemas.
- E.g., attribute *number* for phone number in the instructor schema and for room number in the classroom schema.
- The join of a relation on schema instructor with one on classroom is meaningless.
- While users and application developers can work carefully to ensure use of the right number in each circumstance, having a different attribute name for phone number and for room number serves to reduce user errors.



Naming of Attributes

- While it is a good idea to keep names for incompatible attributes distinct, if attributes of different relations have the same meaning, it may be a good idea to use the same attribute name.
- e.g., using the same attribute name “*name*” for both the instructor and the student entity sets.
- If this was not the case (that is, we used different naming conventions for the instructor and student names), then if we wished to generalize these entity sets by creating a person entity set, we would have to rename the attribute. Thus, even if we did not currently have a generalization of student and instructor, if we foresee such a possibility it is best to use the same name in both entity sets (and relations).



Naming of Attributes

- Although technically, the order of attribute names in a schema does not matter, it is convention to list primary-key attributes first. This makes reading default output (as from select *) easier.



Naming of Relationships

- In large database schemas, relationship sets (and schemas derived therefrom) are often named via a concatenation of the names of related entity sets (with an intervening hyphen or underscore).
 - e.g., *inst stud_dep*
- In some cases, we can use other meaningful names instead of using the longer concatenated names.
 - e.g., *teaches* and *takes*
- We cannot always create relationship-set names by simple concatenation;
 - e.g., for example, *employee_employee* for a manager or works-for relationship between employees
 - Similarly, if there are multiple relationship sets possible between a pair of entity sets, the relationship-set names must include extra parts to identify the relationship set.



Naming of Relationships

- Different organizations have different conventions for naming entity sets.
 - e.g., *student* or *students*
- Using either singular or plural is acceptable, as long as the convention is used consistently across all entity sets.
- As schemas grow larger, with increasing numbers of relationship sets, using consistent naming of attributes, relationships, and entities makes life much easier for the database designer and application programmers.



Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id*, and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
 - faster lookup
 - extra space and extra execution time for updates
 - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as
$$\text{course} \bowtie \text{prereq}$$
 - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors



Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:

Instead of *earnings (company_id, year, amount)*, use

- *earnings_2004, earnings_2005, earnings_2006*, etc., all on the schema (*company_id, earnings*).
 - ▶ Above are in BCNF, but make querying across years difficult and needs new table each year
- *company_year (company_id, earnings_2004, earnings_2005, earnings_2006)*
 - ▶ Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
 - ▶ Is an example of a **crosstab**, where values for one attribute become column names
 - ▶ Used in spreadsheets, and in data analysis tools



Outline

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
 - Functional Dependency Theory
 - Second Normal Form
 - Boyce-Codd Normal Form
 - Third Normal Form
 - Normalization's Goal
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- **Modeling Temporal Data**



Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid*.
- A **snapshot** is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
 - attributes, e.g., address of an instructor at different points in time
 - entities, e.g., time duration when a student entity exists
 - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard



Modeling Temporal Data

- Adding a temporal component results in functional dependencies like
 $ID \rightarrow street, city$
not to hold, because the address varies over time
- A **temporal functional dependency** $X \rightarrow Y$ holds on schema R if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances $r(R)$.
 t



Modeling Temporal Data

- In practice, database designers may add start and end time attributes to relations
 - E.g., $course(course_id, course_title)$ is replaced by
 $course(course_id, course_title, start, end)$

(CS-101, “Introduction to Programming”, 1985-01-01, 2000-12-31)

(CS-101, “Introduction to C”, 2001-01-01, 2010-12-31)

(CS-101, “Introduction to Java”, 2011-01-01, 9999-12-31)

- ▶ Constraint: no two tuples can have overlapping valid times
 - Hard to enforce efficiently



Modeling Temporal Data

- The original primary key for a temporal relation would no longer uniquely identify a tuple. To resolve this problem, we could add the start and end time attributes to the primary key.
- To specify a foreign key referencing such a relation, the referencing tuples would have to include the start- and end-time attributes as part of their foreign key, and the values must match that in the referenced tuple.
- If the system supports temporal data in a better fashion, we can allow the referencing tuple to specify a point in time, rather than a range, and rely on the system to ensure that there is a tuple in the referenced relation whose valid time interval contains the point in time.
 - E.g., student transcript should refer to course information at the time the course was taken



Questions?

Based on the slides of the course book



Database Systems

Lecture 9: Complex Data Types

Dr. Momtazi
momtazi@aut.ac.ir



Outline

- **Semi-Structured Data**
 - XML
 - JSON
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



Semi-Structured Data

- Many applications require storage of complex data, whose schema changes often
- The relational model's requirement of atomic data types may be an overkill
 - E.g., storing set of interests as a set-valued attribute of a user profile may be simpler than normalizing it
- Data exchange can benefit greatly from semi-structured data
 - Exchange can be between applications, or between back-end and front-end of an application
 - Web-services are widely used today, with complex data fetched to the front-end and displayed using a mobile app or JavaScript
- JSON and XML are widely used semi-structured data models



Features of Semi-Structured Data Models

- **Flexible schema**
 - **Wide column** representation: allow each tuple to have a different set of attributes, can add new attributes at any time
 - **Sparse column** representation: schema has a fixed but large set of attributes, by each tuple may store only a subset
- **Multivalued data types**
 - **Sets, multisets**
 - E.g.,: set of interests {'basketball', 'La Liga', 'cooking', 'anime', 'jazz'}
 - **Key-value map** (or just **map** for short)
 - Store a set of key-value pairs
 - E.g., {(brand, Apple), (ID, MacBook Air), (size, 13), (color, silver)}
 - Operations on maps: *put(key, value)*, *get(key)*, *delete(key)*



Features of Semi-Structured Data Models

- **Arrays**
 - Widely used for scientific and monitoring applications
 - E.g., readings taken at regular intervals can be represented as array of values instead of (time, value) pairs
 - [5, 8, 9, 11] instead of {(1,5), (2, 8), (3, 9), (4, 11)}
- Multi-valued attribute types
 - Modeled using *non first-normal-form (NFNF)* data model
 - Supported by most database systems today
- **Array database:** a database that provides specialized support for arrays
 - E.g., compressed storage, query language extensions etc
 - Oracle GeoRaster, PostGIS, SciDB, etc



Nested Data Types

- Hierarchical data is common in many applications
- JSON: JavaScript Object Notation
 - Widely used today
- XML: Extensible Markup Language
 - Earlier generation notation, still used extensively



Motivation

- Each application area has its own set of standards for representing information
- XML/JSON have become the basis for all new generation data interchange formats
- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
 - Similar in concept to email headers
 - Does not allow for nested structures, no standard “type” language
 - Tied too closely to low level document structure (lines, spaces, etc)





Motivation

- Data interchange is critical in today's networked world
 - Examples:
 - 4 Banking: funds transfer
 - 4 Order processing (especially inter-company orders)
 - 4 Scientific data
 - Chemistry: ChemML, ...
 - Genetics: BSML (Bio-Sequence Markup Language), ...
 - Paper flow of information between organizations is being replaced by electronic flow of information



Example

- Purchase orders are typically generated by one organization and sent to another.
- Traditionally they were printed on paper by the purchaser and sent to the supplier; the data would be manually re-entered into a computer system by the supplier.
- This slow process can be greatly sped up by sending the information electronically between the purchaser and supplier.
- XML/JSON provides a standard way of tagging the data.
- The two organizations must of course agree on what tags appear in the purchase order, and what they mean



Motivation for Nesting

- Each purchase order has a purchaser and a list of items as two of its nested structures.
 - Each item in turn has an item identifier, description and a price nested within it.
 - The purchaser has a name and address nested within it.
- Such information are normally split into multiple relations in a relational schema:
 - Item information
 - Purchaser information
 - Purchase orders
 - The relationship between purchase orders, purchasers, and items



Motivation for Nesting

- The relational representation helps to avoid redundancy.
 - E.g., item descriptions would be stored only once for each item identifier in a normalized relational schema.
- In the XML purchase order, however, the descriptions may be repeated in multiple purchase orders that order the same item.
- Gathering all information related to a purchase order into a single nested structure, even at the cost of redundancy, is attractive when information has to be exchanged with external parties.



Applications

- Storing data with complex structures
 - Applications that need to store data that are structured, but are not easily modeled as relations
 - 4 E.g., user preferences stored by a Web browser
 - Editable document representation as part of office application packages for storing documents
 - 4 E.g., Open Office: Open Document Format (ODF)
 - 4 E.g., Microsoft Office: Office Open XML (OOXML)



Applications

- Standardized data exchange formats
 - E.g., ChemML: a standard for representing/exchanging information about chemicals
 - 4 Molecular structure
 - 4 Boiling and melting points
 - 4 Calorific values
 - 4 Solubility in various solvents
 - E.g., required information for business-to-business (B2B) market
 - 4 Product descriptions
 - 4 Price information
 - 4 Product inventories
 - 4 Quotes for a proposed sale



Outline

- Semi-Structured Data
 - **XML**
 - 4 Structure of XML Data
 - 4 XML Document Schema
 - 4 Querying and Transformation
 - 4 Storage of XML Data
 - JSON
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML



Introduction

- Markup: anything in a document that is not intended to be part of the printed output.
- Such notes convey extra information about the text.
 - For example, a typeset in a magazine may want to make notes about how the typesetting should be done. These notes should be distinguished from the actual content, so that they do not end up printed in the magazine.
- In electronic document processing, a markup language is a formal description of what part of the document is content, what part is markup, and what the markup means.



Introduction

- Documents have tags giving extra information about sections of the document
 - E.g.

```
<title> XML </title>
<slide> Introduction ...</slide>
```
- **Extensible**, unlike HTML
 - Users can add new tags, and *separately* specify how the tag should be handled for display



Introduction

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
 - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
 - E.g.

```
<university>
  <department>
    <dept_name> Comp. Sci. </dept_name>
    <building> Taylor </building>
    <budget> 100000 </budget>
  </department>
  <course>
    <course_id> CS-101 </course_id>
    <title> Intro. to Computer Science </title>
    <dept_name> Comp. Sci </dept_name>
    <credits> 4 </credits>
  </course>
</university>
```



Example of XML Data

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser> .... </purchaser>
    <itemlist>
        <item>
            <identifier> RS1 </identifier>
            <description> Atom powered rocket sled </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier> SG2 </identifier>
            <description> Superb glue </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure> we can use different tags in items
            <price> 29.95 </price>
        </item>
    </itemlist>
    <total cost> 429.85 </total cost>
    <payment terms> Cash-on-delivery </payment terms>
</purchase_order>
```



Motivation

- Each XML based standard defines what are valid elements, using
 - XML type specification languages to specify the syntax
 - 4 DTD (Document Type Descriptors)
 - 4 XML Schema
 - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
 - However, this may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data



Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated
- Better than relational tuples as a data-exchange format
 - Unlike relational tuples, XML data is self-documenting due to presence of tags
 - Non-rigid format: tags can be added or ignored
 - E.g., <unit-of-measure>
 - Allows nested structures
 - Wide acceptance, not only in database systems, but also in browsers, tools, and applications



XML Features

- Data Exchange
- Hierarchical Structure
- Extensibility
- Self-Describing
- Integrating XML with Relational Databases



XML Features

- An XML tree starts at a root element and branches from the root to child elements.
- All elements can have sub elements (child elements)

The diagram illustrates the structure of an XML document. On the left, the word "Root element" is written above a large curly brace that spans from the opening tag of the root element to its closing tag. Inside this brace, the XML code is shown. On the right, the word "Child elements" is written above another large curly brace that spans from the opening tag of the first child element to its closing tag. This brace also encloses all other child elements. The XML code is as follows:

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```



XML Features

- XML is extensible, in that element can be easily added as needed.

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <state>ID</state>
  <country>USA</country>
</shipto>
```



XML Features

- XML is self-describing (sort of) with the use of element tags
 - Human-readable format
 - Tags describe the content of the element (sort of)

From reading the tags, it's pretty clear that we're talking about a "Ship To" address that contains the name, address, city & country.

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

But it doesn't provide full metadata, e.g.:

- What's the data type?
- What's the business definition?
- Is <name> a required field?



XML Features

- Similar to DDL, an XML Schema (XSD) defines the structure & format of data

Metadata

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XSD

Data

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

XML

Data

Data

Order Shipment

Ship to:
John Smith
123 Main ST
Boise
USA





XML Features

- XML is often used in conjunction with relational databases for permanent storage and integration with other operational, reporting, and reference data.





Outline

- Semi-Structured Data
 - **XML**
 - 4 **Structure of XML Data**
 - 4 XML Document Schema
 - 4 Querying and Transformation
 - 4 Storage of XML Data
 - JSON
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with `<tagname>` and ending with matching `</tagname>`
- Every document must have a single top-level root element
 - That encompasses all other elements in the document
 - E.g., `<university> ... </university>`
- Elements must be properly **nested**
 - Proper nesting
 - 4 `<course> ... <title> </title> </course>`
 - Improper nesting
 - 4 `<course> ... <title> </course> </title>`
 - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.



Structure of XML Data

- Mixture of text with sub-elements is legal in XML.

- Example:

```
<course>
    This course is being offered for the first time in 2009.
    <course id> BIO-399 </course id>
    <title> Computational Biology </title>
    <dept name> Biology </dept name>
    <credits> 3 </credits>
</course>
```

- Useful for document processing rather than data processing which deals with structured data



Attributes

- Elements can have **attributes**

```
<course course_id= "CS-101">  
    <title> Intro. to Computer Science</title>  
    <dept name> Comp. Sci. </dept name>  
    <credits> 4 </credits>  
</course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course course_id = "CS-101" credits="4">
```



Attributes vs. Subelements

- Distinction between subelement and attribute
 - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
 - In the context of data representation, the difference is unclear and may be confusing
 - 4 Same information can be represented in two ways
 - <course course_id= “CS-101”> ... </course>
 - <course>
 <course_id>CS-101</course_id> ...
 </course>
 - Suggestion: use attributes for identifiers of elements, and use subelements for other contents



Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use `unique-name:element-name`
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">  
    ...  
    <yale:course>  
        <yale:course_id> CS-101 </yale:course_id>  
        <yale:title> Intro. to Computer Science</yale:title>  
        <yale:dept_name> Comp. Sci. </yale:dept_name>  
        <yale:credits> 4 </yale:credits>  
    </yale:course>  
    ...  
</university>
```



More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a `/>` and deleting the end tag
 - `<course course_id="CS-101" Title="Intro. To Computer Science" dept_name = "Comp. Sci." credits="4" />`
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
 - `<![CDATA[<course> ... </course>]]>`
Here, `<course>` and `</course>` are treated as just strings
CDATA stands for “character data”



Outline

- Semi-Structured Data
 - **XML**
 - 4 Structure of XML Data
 - 4 **XML Document Schema**
 - 4 Querying and Transformation
 - 4 Storage of XML Data
 - JSON
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
 - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
 - **Document Type Definition (DTD)**
 - 4 Widely used
 - **XML Schema**
 - 4 Newer, increasing use



Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
 - What elements can occur
 - What attributes can/must an element have
 - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types like integer ...
 - All values represented as strings in XML
- DTD syntax
 - `<!ELEMENT element (subelements-specification) >`
 - `<!ATTLIST element (attributes) >`



Element Specification in DTD

- Subelements can be specified as
 - names of elements, or
 - #PCDATA (parsed character data), i.e., character strings
 - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

```
<! ELEMENT department (dept_name building, budget)>
<! ELEMENT dept_name (#PCDATA)>
<! ELEMENT building(#PCDATA)>
<! ELEMENT budget (#PCDATA)>
```



University DTD

```
<!DOCTYPE university [  
    <!ELEMENT university ( (department|course|instructor|teaches)+)>  
    <!ELEMENT department ( dept name, building, budget)>  
    <!ELEMENT course ( course id, title, dept name, credits)>  
    <!ELEMENT instructor (IID, name, dept name, salary)>  
    <!ELEMENT teaches (IID, course id)>  
    <!ELEMENT dept name( #PCDATA )>  
    <!ELEMENT building( #PCDATA )>  
    <!ELEMENT budget( #PCDATA )>  
    <!ELEMENT course id ( #PCDATA )>  
    <!ELEMENT title ( #PCDATA )>  
    <!ELEMENT credits( #PCDATA )>  
    <!ELEMENT IID( #PCDATA )>  
    <!ELEMENT name( #PCDATA )>  
    <!ELEMENT salary( #PCDATA )>  
]>
```



Element Specification in DTD

- Subelement specification may have regular expressions

```
<!ELEMENT university ( ( department | course | instructor | teaches )+)>
```

4 Notation:

- “|” - or
- “+” - 1 or more occurrences
- “*” - 0 or more occurrences
- “?” - 0 or 1 occurrence



Attribute Specification in DTD

- Attribute specification : for each attribute
 - Name
 - Type of attribute
 - 4 CDATA
 - 4 ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
 - Whether
 - 4 mandatory (#REQUIRED)
 - has a default value (value),
 - 4 or neither (#IMPLIED)
- Examples
 - `<!ATTLIST course course_id CDATA #REQUIRED>`, or
 - `<!ATTLIST course`
`course_id ID #REQUIRED`
`dept_name IDREF #REQUIRED`
`instructors IDREFS #IMPLIED >`



IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
 - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of another element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document



University DTD with Attributes

```
<!DOCTYPE university-3 [  
    <!ELEMENT university ( (department|course|instructor)+)>  
    <!ELEMENT department ( building, budget )>  
    <!ATTLIST department  
        dept_name ID #REQUIRED >  
    <!ELEMENT course (title, credits )>  
    <!ATTLIST course  
        course_id ID #REQUIRED  
        dept_name IDREF #REQUIRED  
        instructors IDREFS #IMPLIED >  
    <!ELEMENT instructor ( name, salary )>  
    <!ATTLIST instructor  
        IID ID #REQUIRED  
        dept_name IDREF #REQUIRED >  
    ...  
>]
```



XML data with ID and IDREF attributes

```
<university-3>
    <department dept name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id="CS-101" dept name="Comp. Sci" instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    ...
    <instructor IID="10101" dept name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    ...
</university-3>
```



Limitations of DTDs

- No typing of text elements and attributes
 - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
 - $(A \mid B)^*$ allows specification of an unordered set
 - 4 Cannot ensure the number of times each of A and B occurs
- IDs and IDREFs are untyped
 - The *instructors* attribute of an course may contain a reference to a department or another course, which is meaningless
 - 4 *instructors* attribute should ideally be constrained to refer to instructor elements



XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs.
- Supports typing
 - Built-in types
 - 4 E.g., string, integer, decimal, and boolean
 - 4 Also, constraints on min/max values
 - User-defined types
 - 4 Simple types with added restrictions
 - 4 Complex types constructed using constructors
- Many more features, including
 - 4 uniqueness and foreign key constraints, inheritance



XML Schema

- XML Schema is itself specified in XML syntax, unlike DTDs
 - More-standard representation, but verbose
- XML Scheme is integrated with namespaces; E.g., “xs:”
- BUT: XML Schema is significantly more complicated than DTDs.



XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="dept name" type="xs:string"/>
      <xs:element name="building" type="xs:string"/>
      <xs:element name="budget" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
.....
<xs:element name="instructor">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="IID" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="dept name" type="xs:string"/>
      <xs:element name="salary" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
... Contd.
```



XML Schema Version of Univ. DTD (Cont.)

```
....  
<xs:complexType name="UniversityType">  
  <xs:sequence>  
    <xs:element ref="department" minOccurs="0" maxOccurs="unbounded"/>  
    <xs:element ref="course" minOccurs="0" maxOccurs="unbounded"/>  
    <xs:element ref="instructor" minOccurs="0" maxOccurs="unbounded"/>  
    <xs:element ref="teaches" minOccurs="0" maxOccurs="unbounded"/>  
  </xs:sequence>  
</xs:complexType>  
</xs:schema>
```

- Choice of “xs:” was ours -- any other namespace prefix could be chosen
- Element “university” has type “universityType”, which is defined separately
 - xs:complexType is used later to create the named complex type “UniversityType”



More features of XML Schema

- Attributes specified by xs:attribute tag:
 - `<xs:attribute name = "dept_name"/>`
 - adding the attribute `use = "required"` means value must be specified



More features of XML Schema

- Key constraint: “department names form a key for department elements under the root university element:

```
<xs:key name = “deptKey”>  
    <xs:selector xpath = “/university/department”/>  
    <xs:field xpath = “dept_name”/>  
<\xs:key>
```

- Foreign key constraint from course to department:

```
<xs:keyref name = “courseDeptFKey” refer=“deptKey”>  
    <xs:selector xpath = “/university/course”/>  
    <xs:field xpath = “dept_name”/>  
<\xs:keyref>
```



XML Schema vs DTD

- Constraining elements texts to specific types; e.g., numeric types or complex types
- User-defined types
- Uniqueness and foreign-key constraints.
- Integrating with namespaces to allow different parts of a document to conform to different schemas.
- Restricting types to create specialized types, for instance by specifying minimum and maximum values.
- Complex types to be extended by using a form of inheritance.



Outline

- Semi-Structured Data
 - **XML**
 - 4 Structure of XML Data
 - 4 XML Document Schema
 - 4 **Querying and Transformation**
 - 4 Storage of XML Data
 - JSON
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



Querying and Transforming XML Data

- Main applications
 - Extract information from large bodies of XML data
 - Converting data between different representations in XML
- Main requirement: tools for
 - Translation of information from one XML schema to another
 - Querying on XML data
- Above two are closely related, and handled by the same tools



Querying and Transforming XML Data

- Standard XML querying/translation languages
 - XPath
 - 4 Simple language consisting of path expressions
 - 4 a building block for XQuery
 - XQuery
 - 4 An XML query language with a rich set of features
 - 4 It is modeled after SQL but is significantly different, since it has to deal with nested XML data.
 - 4 XQuery also incorporates XPath expressions.
 - XSLT
 - 4 Simple language designed for translation from XML to XML and XML to HTML
 - 4 It is used primarily in document-formatting applications, rather than data management applications



Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
 - Element nodes have child nodes, which can be attributes or subelements
 - Text in an element is modeled as a text node child of the element
 - Children of a node are ordered according to their order in the XML document
 - Element and attribute nodes (except for the root node) have a single parent, which is an element node
 - The root node has a single child, which is the root element of the document



XPath

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by “/”
 - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
 - E.g. /university-3/instructor/name returns
`<name>Srinivasan</name>`
`<name>Brandt</name>`
 - E.g. /university-3/instructor/name/text() returns
Srinivasan
Brandt
- The nodes returned by each step appear in the same order as their appearance in the document.



XPath

- The initial “/” denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
 - Each step operates on the set of instances produced by the previous step
- Attribute values are accessed, using the “@” symbol.
 - E.g. `/university-3/course/@course_id` returns a set of all values of course_id attributes of course elements
- IDREF attributes are not dereferenced automatically (more on this later)



XPath

- Selection predicates may follow any step in a path, in []
 - E.g. /university-3/course[credits] returns
4 course elements containing a credits subelement
 - E.g. /university-3/course[credits >= 4] returns
4 course elements with a credits value greater than or equal to 4,
 - E.g. /university-3/course[credits >= 4]/@course_id returns
4 returns the course identifiers of courses with credits >= 4



Functions in XPath

- XPath provides several functions
- The function `count()` at the end of a path counts the number of elements in the set generated by the path
 - E.g. `/university-2/instructor[count(.//teaches/course)> 2]` returns 4 instructors teaching more than 2 courses (on university-2 schema)
- Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives `and` and `or` and function `not()` can be used in predicates



Functions in XPath

- The function `id("k")` returns the node (if any) with an attribute of type ID and value “k”.
- IDREFs can be referenced using function `id()`
 - `id()` can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
 - E.g. `/university-3/course/id(@dept_name)` returns
 - 4 all department elements referred to from the dept_name attribute of course elements.
 - E.g. `/university-3/course/id(@instructors)` returns
 - 4 all instructor elements referred to from the instructor attribute of course elements.



More XPath Features

- Operator “|” used to implement union
 - E.g. `/university-3/course[@dept name=“Comp. Sci”] | /university-3/course[@dept name=“Biology”]`
 - 4 Gives union of Comp. Sci. and Biology courses
 - 4 However, “|” cannot be nested inside other operators.
 - 4 The nodes in the union are returned in the order in which they appear in the document



More XPath Features

- `doc(name)` returns the root of a named document
- Thus, a path expression can be applied on a specified document, instead of being applied on the current default document.
 - E.g. `doc("university.xml")/university/department` returns all departments at the university
- The function `collection(name)` is similar to `doc`, but returns a collection of documents identified by name



More XPath Features

- “//” can be used to skip multiple levels of nodes
 - E.g. `/university-3//name`
 - 4 finds any `name` element *anywhere* under the `/university-3` element, regardless of the element in which it is contained.
- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
 - “//”, described above, is a short form for specifying “all descendants”
 - “..” specifies the parent.



XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
- XQuery queries are modeled after SQL queries, but differ significantly from SQL .



XQuery

- XQuery is organized into five sections
for ... let ... where ... order by ...return ...
- XQuery syntax
 - for** ⇔ SQL **from**
 - let** allows temporary variables, and has no equivalent in SQL
 - where** ⇔ SQL **where**
 - order by** ⇔ SQL **order by**
 - return** ⇔ SQL **select** (allows the construction of results in XML)
- It is referred to as “ FLWOR ” (pronounced “flower”) expression
- A FLWOR query need not contain all the clauses



FLWOR Syntax in XQuery

- Simple FLWOR expression in XQuery
 - find all courses with credits > 3, with each result enclosed in an `<course_id> .. </course_id>` tag

```
for $x in /university-3/course
let $courseld := $x/@course_id
where $x/credits > 3
return <course_id> { $courseld } </course id>
```



FLWOR Syntax in XQuery

```
for $x in /university-3/course
let $courseld := $x/@course_id
where $x/credits > 3
return <course_id> { $courseld } </course id>
```

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- The where clause, like the SQL where clause, performs additional tests on the joined tuples from the for clause
- Since the for clause uses XPath expressions, selections may occur within the XPath expression.

```
for $x in /university-3/course[credits > 3]
```



FLWOR Syntax in XQuery

```
for $x in /university-3/course  
let $courseld := $x/@course_id  
where $x/credits > 3  
return <course_id> { $courseld } </course id>
```

- The let clause simply allows the results of XPath expressions to be assigned to variable names for simplicity of representation
- Let clause not really needed in this query
- In fact, since this query is simple, we can easily do away with the let clause, and the variable **\$courseld** in the return clause could be replaced with **\$x/@course id**.

```
for $x in /university-3/course[credits > 3]  
return <course_id> { $x/@course_id } </course_id>
```



FLWOR Syntax in XQuery

```
for $x in /university-3/course
let $courseld := $x/@course_id
where $x/credits > 3
return <course_id> { $courseld } </course id>
```

- Items in the return clause are XML text unless enclosed in {}, in which case they are evaluated as expressions
- The query can be modified to return an element with tag course, with the course identifier as an attribute, by replacing the return clause with the following:

```
return <course course id="{$x/@course id}" />
```



FLWOR Syntax in XQuery

```
for $x in /university-3/course
let $courseld := $x/@course_id
where $x/credits > 3
return <course_id> { $courseld } </course id>
```

- XQuery also supports constructing elements using the element and attribute constructors.

- E.g.,

```
return element course {
    attribute course_id {$x/@course_id},
    attribute dept_name {$x/dept_name},
    element title {$x/title},
    element credits {$x/credits}
}
```

returns course elements with course_id and dept_name as attributes and title and credits as subelements.



Other XQuery Abilities

- Join
- Nested queries
- Grouping and aggregation
- Sorting
- Interface to application programs (Java DOM API)



XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
 - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
 - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
 - Templates combine selection using XPath with construction of results



Outline

- Semi-Structured Data
 - **XML**
 - 4 Structure of XML Data
 - 4 XML Document Schema
 - 4 Querying and Transformation
 - 4 **Storage of XML Data**
 - JSON
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



Storage of XML Data

- XML data can be stored in
 - Non-relational data stores
 - 4 Flat files
 - Natural for storing XML
 - But has all problems discussed in Chapter 1
 - » Data isolation
 - » Atomicity
 - » Concurrent access
 - » Security
 - 4 XML database
 - Database built specifically for storing XML data, supporting DOM model and declarative querying
 - Currently no commercial-grade systems



Storage of XML Data

- XML data can be stored in
 - Relational databases
 - 4 Data must be translated into relational form
 - 4 Advantage: mature database systems
 - 4 Disadvantages: overhead of translating data and queries



Storage of XML in Relational Databases

- Approaches:
 - String Representation
 - Tree Representation
 - Map to relations



String Representation

- Store each top level element as a string field of a tuple in a relational database
 - Use a single relation to store all elements
 - 4 Easy to use, but no knowledge about the schema of the stored elements
 - 4 Not possible to query the data directly
 - Use a separate relation for each top-level element type
 - 4 E.g. department, course, instructor, and teaches
 - Each with a string-valued attribute to store the element
- Indexing:
 - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
 - 4 E.g. dept_name, course id, or name



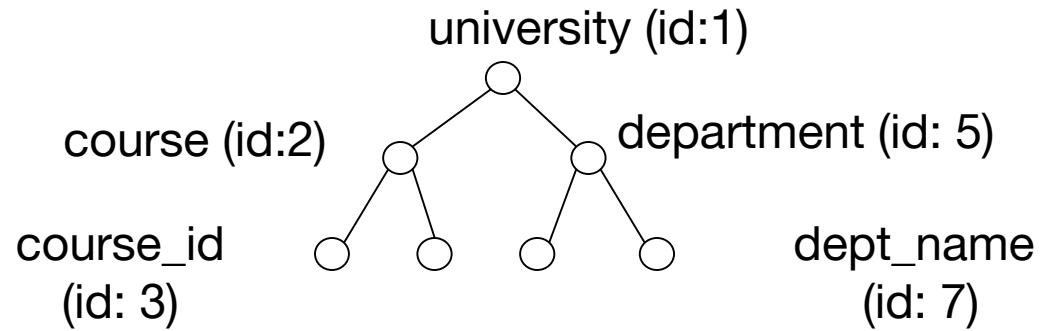
String Representation

- Benefits:
 - Can store any XML data even without DTD
 - As long as there are many top-level elements in a document, strings are small compared to full document
 - 4 Allows fast access to individual elements.
- Drawback: Need to parse strings to access values inside the elements
 - Parsing is slow.



Tree Representation

- **Tree representation:** model XML data as tree and store using relations
nodes(id, parent_id, type, label, value)



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children



Tree Representation

- Benefit:
 - Can store any XML data, even without DTD
- Drawbacks:
 - Data is broken up into too many pieces, increasing space overheads
 - Even simple queries require a large number of joins, which can be slow



Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
 - An id attribute to store a unique id for each element
 - A relation attribute corresponding to each element attribute
 - A parent_id attribute to keep track of parent element
 - 4 As in the tree representation
 - 4 Position information (i^{th} child) can be stored too
- All subelements that occur only once can become relation attributes
 - For text-valued subelements, store the text as attribute value
 - For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
 - Similar to handling of multivalued attributes when converting ER diagrams to tables



Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get

department(id, dept_name, building, budget)

course(parent_id, course_id, dept_name, title, credits)



Publishing and Shredding XML Data

- **Publishing**: process of converting relational data to an XML format for export to other applications.
- **Shredding**: process of converting back an XML document into a set of tuples to be inserted into one or more relations
- *XML-enabled* database systems support automated publishing and shredding
- Otherwise, application code can perform the publishing and shredding operations



Publishing and Shredding XML Data

- An XML-enabled database supports automatic publishing using map to relation mechanism
- Approaches:
 - Simple
 - 4 Creates an XML element for every row of a table
 - 4 Makes each column in that row a subelement of the XML element
 - Complex
 - 4 Allows nested structures to be created
 - 4 Extensions of SQL with nested queries in the select clause have been developed to allow easy creation of nested XML output



SQL/XML

- New standard SQL extension that allows creation of nested XML output
 - Each output tuple is mapped to an XML element *row*
 - Each relation attribute mapped to an XML element of the same name



SQL/XML

```
<university>
  <department>
    <row>
      <dept name> Comp. Sci. </dept name>
      <building> Taylor </building>
      <budget> 100000 </budget>
    </row>
    <row>
      <dept name> Biology </dept name>
      <building> Watson </building>
      <budget> 90000 </budget>
    </row>
  </department>
  <course>
    <row>
      <course id> CS-101 </course id>
      <title> Intro. to Computer Science </title>
      <dept name> Comp. Sci </dept name>
      <credits> 4 </credits>
    </row>
  </course>
</university>
```



SQL Extensions

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

```
select xmlelement (name "course",
    xmlattributes (course id as course id, dept name as dept name),
    xmlelement (name "title", title),
    xmlelement (name "credits", credits))
from course
```



SQL Extensions

- Xmlagg creates a forest of XML elements
- Creating an element for each department with a course, containing as subelements all the courses in that department.
- Since the query has a clause group by dept_name, the aggregate function is applied on all courses in each department, creating a sequence of course id elements.

```
select xmlelement (name "department", dept_name,
    xmlagg (xmlforest(course_id)
            order by course_id))
from course
group by dept_name
```



Outline

- Semi-Structured Data
 - XML
 - **JSON**
 - RDF
- Object Orientation
- Textual Data
- Spatial Data



What is JSON

- **JSON** (JavaScript Object Notation) is a minimal, readable format for structuring data.
- It is used primarily to transmit data between a server and web application, as an alternative to XML.
- JSON is ubiquitous in data exchange today
 - Widely used for web services
 - Most modern applications are architected around web services
- JSON is verbose
 - Compressed representations such as BSON (Binary JSON) used for efficient data storage



JSON vs. XML

- It is similar to XML in that it is:
 - "self describing" & human readable
 - hierarchical
 - simple & interoperable

- It differs from XML in that it is:
 - can be parsed with standard JavaScript notation
 - uses arrays
 - can be simpler & shorter to read & write
 - parsing is faster



JSON vs. XML

JSON

```
{"employees": [  
    {"firstName": "Shannon", "lastName": "Kempe"},  
    {"firstName": "Anita", "lastName": "Kress"},  
    {"firstName": "Tony", "lastName": "Shaw"}  
]}
```

XML

```
<employees>  
    <employee>  
        <firstName>Shannon</firstName>  
        <lastName>Kempe</lastName>  
    </employee>  
    <employee>  
        <firstName>Anita</firstName>  
        <lastName>Kress</lastName>  
    </employee>  
    <employee>  
        <firstName>Tony</firstName>  
        <lastName>Shaw</lastName>  
    </employee>  
</employees>
```



JSON Metadata

- The JSON schema offers a richer set of metadata.

Data

```
{  
  "id": 127849,  
  "brand": "Super Cooler",  
  "price": 12.50,  
  "tags": ["camping", "sports"]  
}
```

Example Product in the API

Context Needed (i.e. Metadata)

- Can the ID contain letters?
- What is a brand?
- Is a price required?
- Etc.

Metadata

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "Product",  
  "description": "A retail product from Acme's online catalog",  
  "type": "object",  
  "properties": {  
    "id": {  
      "description": "The unique identifier for a product",  
      "type": "integer"  
    },  
    "brand": {  
      "description": "The brand name of the product as shown in the online catalogue",  
      "type": "string"  
    },  
    "price": {  
      "type": "number",  
    },  
    "tags": {  
      "type": "array",  
      "items": {  
        "type": "string"  
      },  
      "minItems": 1,  
    },  
    "required": ["id", "brand", "price"]  
  }  
}
```

JSON Schema



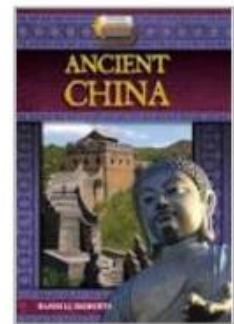
Integrating JSON with Document Databases

- JSON is often used with document databases, such as MongoDB, which uses JSON documents in order to store records
- Document databases are popular ways to store unstructured information in a flexible way (e.g. multimedia, social media posts, etc.)
 - Each Collection can contain numerous Documents which could all contain different fields.

```
{type: "Artifact",  
medium: "Ceramic"  
country: "China",  
}
```



```
{type: "Book",  
title: "Ancient China"  
country: "China",  
}
```





JSON

- Textual representation widely used for data exchange
- Example of JSON data

```
{  
    "ID": "22222",  
    "name": {  
        "firstname": "Albert",  
        "lastname": "Einstein"  
    },  
    "deptname": "Physics",  
    "children": [  
        {"firstname": "Hans", "lastname": "Einstein"},  
        {"firstname": "Eduard", "lastname": "Einstein"}  
    ]  
}
```

- Types: integer, real, string, and
 - *Objects*: are key-value maps, i.e. sets of (attribute name, value) pairs
 - Arrays are also key-value maps (from offset to value)



JSON Structure

- JSON uses map data structure rather than tree data structure
- Syntax follows JS syntax
- data is in name/value pairs
- data is separated by commas
- curly braces { } hold objects
- square brackets [] hold arrays



JSON

- SQL extensions for
 - JSON types for storing JSON data
 - Extracting data from JSON objects using path expressions
 - E.g. $V \rightarrow ID$, or $v.ID$
 - Generating JSON from relational data
 - E.g. `json.build_object('ID', 12345, 'name', 'Einstein')`
 - Creation of JSON collections using aggregation
 - E.g. `json_agg` aggregate function in PostgreSQL
 - Syntax varies greatly across databases



Outline

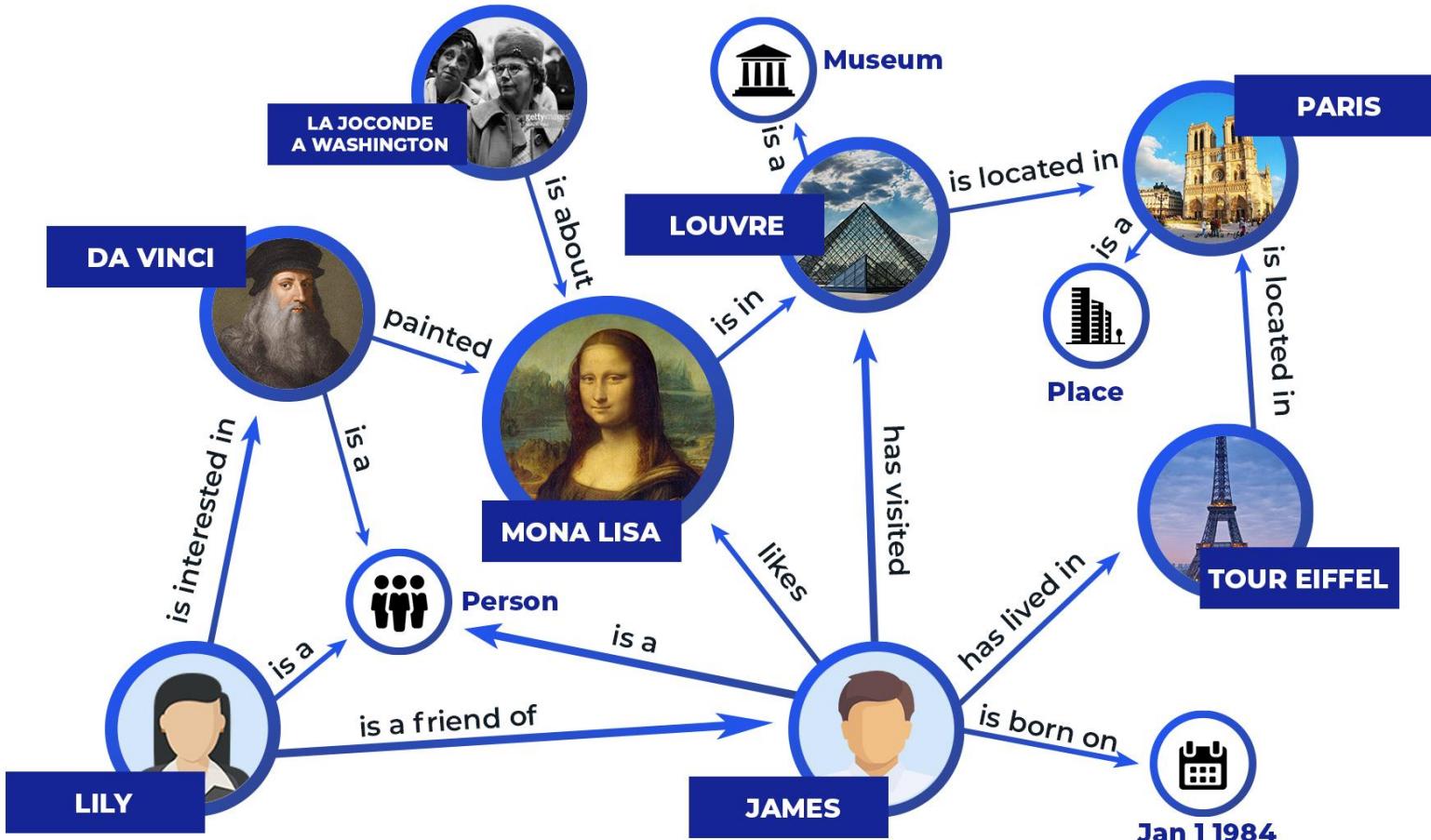
- Semi-Structured Data
 - XML
 - JSON
 - **RDF**
 - 4 Linked Open Data**
 - 4 RDF Structure**
 - 4 SPARQL**
- Object Orientation
- Textual Data
- Spatial Data



Web of Linked Data

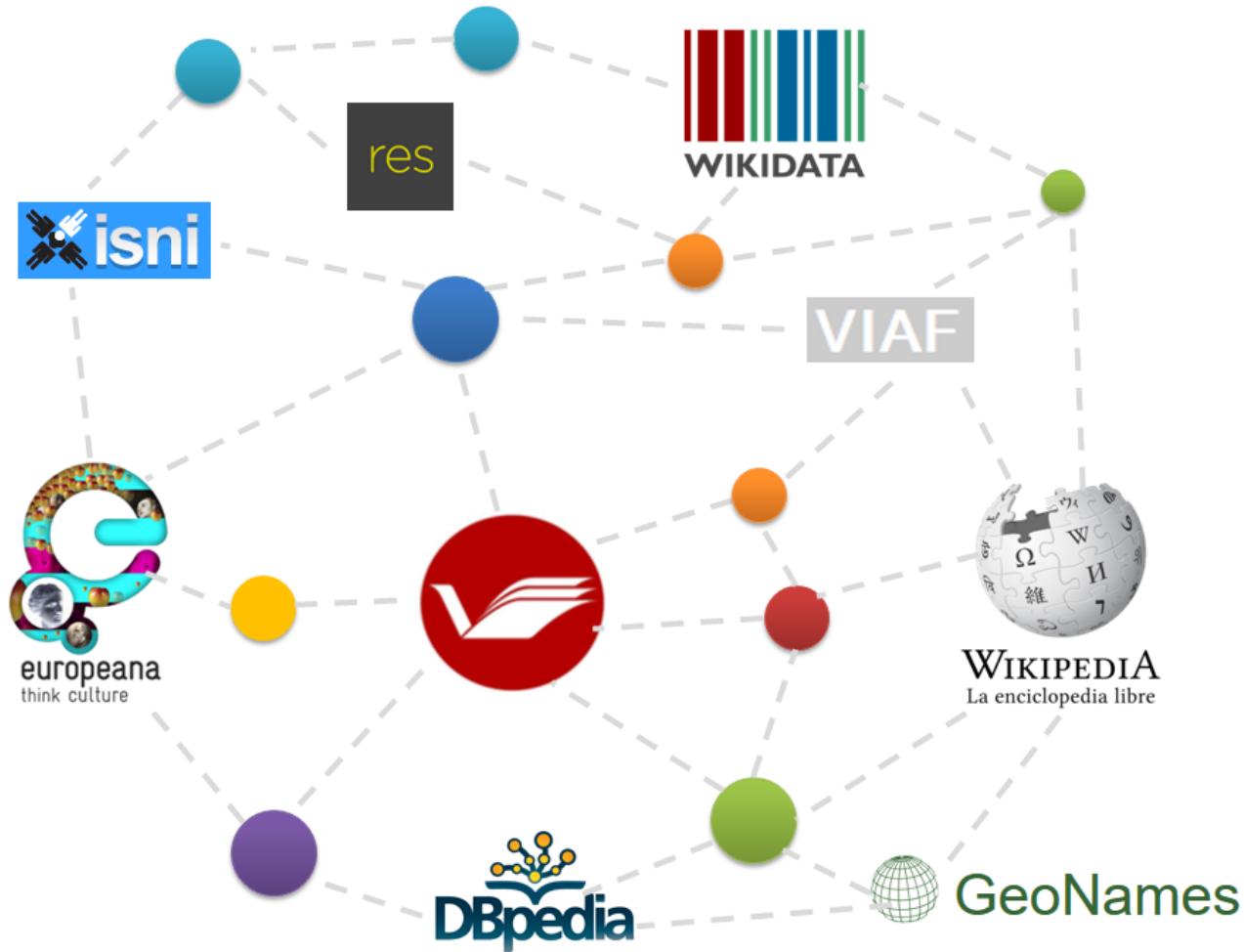
- The Web is evolving from a “Web of linked documents” into a “Web of linked data”
 - The Web started as a collection of documents published online – accessible at a Web location identified by a URL.
 - These documents often contain data about real-world resources which is mainly human-readable and cannot be understood by machines.
 - The Web of Data is about enabling the access to this data, by making it available in machine-readable formats and connecting it using Uniform Resource Identifiers (URIs), thus enabling people and machines to collect the data, and put it together to do all kinds of things with it (permitted by the licence).

Knowledge Graph



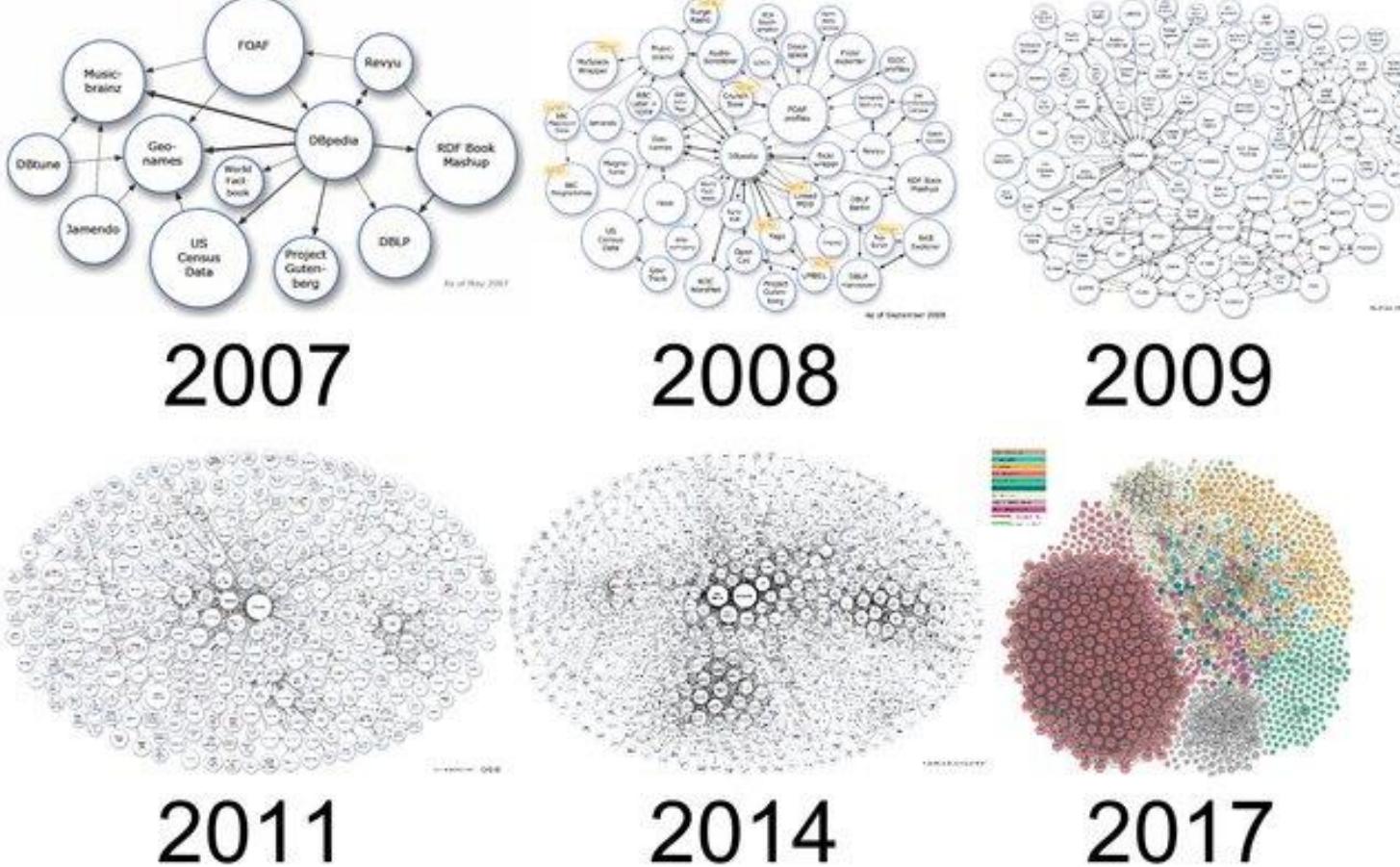


Linked Open Data





Linked Open Data





Defining linked data

- The **four design principles** of Linked Data:
 - Use Uniform Resource Identifiers (URIs) as names for things.
 - Use HTTP URIs so that people can look up those names.
 - When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
 - Include links to other URIs so that they can discover more things.



Design Principles

1. Use Uniform Resource Identifiers (URIs) as names for things.
2. Use HTTP URIs so that people can look up those names.

E.g. for an organisation: UNICEF in EuroVoc.

4 <http://eurovoc.europa.eu/1022>

This site is part of 

 **EuroVoc** Multilingual Thesaurus of the European Union

Europa > EuroVoc homepage > Domains and MT > Unicef

Content language: (en) English ▾

Simple search

Advanced search

Browse

Browse the subject-oriented version

Download

- By domain
- Permutted alphabetical
- Multilingual list
- Alphabetical index
- EuroVoc SKOS/RDF

Unicef

UF United Nations Children's Fund
United Nations International Children's Emergency Fund

76 INTERNATIONAL ORGANISATIONS

MT 7606 United Nations
BT1 UN programmes and funds
RT child [2816]

URI <http://eurovoc.europa.eu/1022>

Has Exact Match

UNICEF = United Nations Children's Fund (ECLAS)

LANGUAGE EQUIVALENTS

BG	УНИЦЕФ
ES	Unicef
CS	Unicef
DA	Unicef
DE	Unicef
ET	Unicef
EL	Unicef
EN	Unicef
FR	Unicef
HR	UNICEF
IT	Unicef
LV	Unicef
LT	Unicef
HU	Unicef
MT	Unicef
NL	Unicef
nl	Unicef



Design Principles

3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs so that people/machines can discover more things.

```
<rdf:Description rdf:about="http://eurovoc.europa.eu/223339">
  <rdf:type rdf:resource="http://eurovoc.europa.eu/schema#PreferredTerm"/>
  <eu:termReleasedWithVersion>n/a</eu:termReleasedWithVersion>
  <xl:literalForm xml:lang="en">Unicef</xl:literalForm>
  <skos:inScheme rdf:resource="http://eurovoc.europa.eu/100285"/>
  <xl:altLabel rdf:resource="http://eurovoc.europa.eu/223340"/>
  <xl:altLabel rdf:resource="http://eurovoc.europa.eu/223341"/>
</rdf:Description>

<rdf:Description rdf:about="http://eurovoc.europa.eu/1022">
  <rdf:type rdf:resource="http://eurovoc.europa.eu/schema#ThesaurusConcept"/>
  <xl:prefLabel rdf:resource="http://eurovoc.europa.eu/223339"/>
  <skos:broader rdf:resource="http://eurovoc.europa.eu/4365"/>
  <skos:related rdf:resource="http://eurovoc.europa.eu/758"/>
  <skos:exactMatch rdf:resource="http://ec.europa.eu/eclas/euc11/000005663"/>
</rdf:Description>
```



Linked Data vs. Open Data

“Open data is data that can be freely used, reused and redistributed by anyone – subject only, at most, to the requirement to attribute and share-alike.”

- OpenDefinition.org

Open data

Data can be published and be publicly available under an open licence without linking to other data sources.



Linked data

Data can be linked to URLs from other data sources, using open standards such as RDF without being publicly available under an open licence.



5 star-schema of Linked Open Data

★	Make your stuff available on the Web (whatever format) under an open license.
★★	Make it available as structured data (e.g., Excel instead of image scan of a table).
★★★	Use non-proprietary formats (e.g., CSV instead of Excel).
★★★★	Use URIs to denote things, so that people can point at your stuff.
★★★★★	Link your data to other data to provide context.



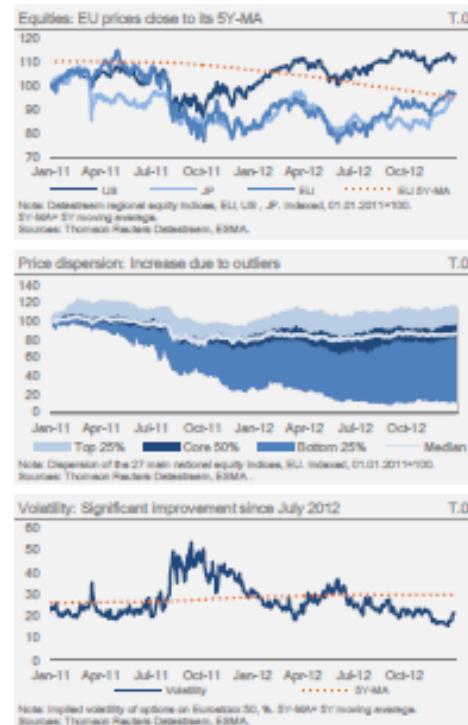
5 star-schema of Linked Open Data

- ★ Make your stuff available on the Web under an open licence



Securities markets

Equity markets



Equity markets developed positively in 2012. After a volatile first semester, equity indices increased as from early August 2012, along with an overall improvement of financial markets in the EU, following the announcement of Outright Monetary Transactions (OMT) by the ECB and further progress on the establishment of a Banking Union. The improvement can also be seen in liquidity and volatility indicators, which performed better than their five year averages.

Global Equities: After bottoming out in July, the EU equity index increased by 10% as from then, edging closer to its five-year average. Compared to Japan, EU indices performed slightly better as from July 2012. However, EU indices underperformed compared to the US, due to macroeconomic conditions. US equity markets were around 12% higher than their early 2011 level, while European and Japanese indices were still 3% lower.

Dispersion: As from August, most of the EU national equity indices experienced increases. More precisely, from then on the top 75% followed an upward trend. However, one European country suffered a very significant decline from January 2011, as indicated by the lowest value of the bottom 25%. The aggregate effect of this sharp decline was mitigated by the fact that the country in question is relatively small in terms of market capitalisation. However, it does indicate European equity markets' increasing differentiation, as price movements in national equity indices for the bottom quartile are decoupled from the generally positive trends in other national indices.

Volatility: Expected volatility, measured by the implied volatility of options on the Eurostoxx 50, decreased from July 2012 and at year-end was around 8.5 percentage points lower than its five-year average. Having been



5 star-schema of Linked Open Data

★ ★ Make it available as structured data

A	B	C
Value	Definition	PortDescription
1	Albania	
2 AL	Austria	
3 AT	Bosnia and Herzegovina	
4 BA	Belgium	
5 BE	Bulgaria	
6 BG	Switzerland	
7 CH	Cyprus	
8 CY	Czech Republic	
9 CZ	Germany	
10 DE	Denmark	
11 DK	Estonia	
12 EE	Spain	
13 ES	Finland	
14 FI	France	
15 FR	United Kingdom	
16 GB	Greece	
17 GR	Croatia	
18 HR	Hungary	
19 HU	Ireland	
20 IE	Iceland	
21 IS	Italy	
22 IT	Liechtenstein	
23 LI	Lithuania	
24 LT	Luxembourg	
25 LU	Latvia	
26 LV	Montenegro	
27 ME	Macedonia, the Former Yugoslav Republic of	
28 MK	Malta	
29 MT	Netherlands	
30 NL	Norway	
31 NO	Poland	
32 PL	Portugal	
33 PT	Romania	
34 RO	Serbia	
35 RS	Sweden	
36 SE	Slovenia	
37 SI		



5 star-schema of Linked Open Data

★ ★ ★ Use non-proprietary formats

- Proprietary: Excel, Word, PDF...
- Non-proprietary: XML, CSV, RDF, JSON, ODF...

```
<?xml version="1.0"?>
<iat:activities generated-datetime="2014-01-09T13:23:20" version="1.0">
  - <iat:activity xml:lang="en" last-updated-datetime="2014-01-09T13:13:09" default-currency="EUR">
    <reporting-org type="15" ref="EC-ELARG">European Commission - Enlargement</reporting-org>
    <iat:identifier>EU-2010/22028/1</iat:identifier>
    <other-identifier owner-name="European Commission - Enlargement" owner-ref="EC-ELARG">2010/22028/1</other-identifier>
    <title>Support for Improvement in Governance and Management (SIGMA) in the Western Balkans and Turkey </title>
    <description>2010 Multi-Beneficiary Programme under the IPA Transition Assistance and Institution Building Component - part implemented by DG ELARG (other parts encoded under 2009/022-029 (DG ENTR) and 2009/022-030 (DG TAXUD)). Support for Improvement in Governance and Management (SIGMA) in the Western Balkans and Turkey </description>
    <activity-status code="2">Implementation</activity-status>
    <activity-date type="start-planned" iso-date="2010-12-20">2010-12-20</activity-date>
    <participating-org role="Funding" type="15" ref="EC-ELARG">European Commission - Enlargement</participating-org>
    <participating-org role="Extending" type="15" ref="EU-1">Commission of the European Communities</participating-org>
    <participating-org role="Implementing" ref="50000">OTHER</participating-org>
    <recipient-region code="89">Europe, regional</recipient-region>
    <sector code="15110" vocabulary="DAC">Public sector policy and administrative management</sector>
    <policy-marker code="01" vocabulary="DAC" significance="0">Gender Equality</policy-marker>
    <policy-marker code="02" vocabulary="DAC" significance="0">Aid to Environment</policy-marker>
    <policy-marker code="03" vocabulary="DAC" significance="2">Participatory Development/Good</policy-marker>
    <policy-marker code="04" vocabulary="DAC" significance="0">Trade Development</policy-marker>
    <policy-marker code="05" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Convention on Biological Diversity</policy-marker>
    <policy-marker code="06" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Framework Convention on Climate Change - Mitigation</policy-marker>
    <policy-marker code="07" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Framework Convention on Climate Change - Adaptation</policy-marker>
    <policy-marker code="08" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Convention to Combat Desertification</policy-marker>
    <collaboration-type code="1">Bilateral</collaboration-type>
    <default-flow-type code="10">ODA (Official development Assistance)</default-flow-type>
    <default-finance-type code="110">Grant</default-finance-type>
    <default-aid-type code="C01">Project-type interventions</default-aid-type>
  - <transaction ref="EU-2010/22028/1/0">
    <transaction-type code="C">Commitment</transaction-type>
    <value>10000000</value>
    <transaction-date iso-date="2010-05-06">2010-05-06</transaction-date>
  </transaction>
  - <transaction ref="EU-2010/22028/13/0">
    <transaction-type code="C">Commitment</transaction-type>
    <value>1500000</value>
    <transaction-date iso-date="2010-05-06">2010-05-06</transaction-date>
  </transaction>
  - <transaction ref="EU-2010/22028/15/0">
```



5 star-schema of Linked Open Data

★ ★ ★ ★ Use URIs to denote things

```
<rdf:Description rdf:about="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/additive-300">
  <additives:id>300</additives:id>
  <additives:additiveName>Oxidized polyethylene wax</additives:additiveName>
  <additives:isGroup>No</additives:isGroup>
  <additives:eNumber>E 914</additives:eNumber>
  <rdf:type rdf:resource="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/Additive"/>
</rdf:Description>
<rdf:Description rdf:about="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/additive-301">
  <additives:eNumber>E 920</additives:eNumber>
  <additives:additiveName>L-cysteine</additives:additiveName>
  <additives:isGroup>No</additives:isGroup>
  <rdf:type rdf:resource="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/Additive"/>
  <additives:id>301</additives:id>
</rdf:Description>
```



5 star-schema of Linked Open Data

★ ★ ★ ★ ★ Link your data to other data to provide context

```
<skos:Concept rdf:about="http://publications.europa.eu/resource/authority/corporate-body/UNICEF"
  at:deprecated="false">
  <skos:inScheme rdf:resource="http://publications.europa.eu/resource/authority/corporate-body"/>
  <skos:broader rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNO"/>
  <at:authority-code>UNICEF</at:authority-code>
  <at:op-code>UNICEF</at:op-code>
  <atold:op-code>UNICEF</atold:op-code>
  <dc:identifier>UNICEF</dc:identifier>
  <at:start.use>1951-01-01</at:start.use>
  <skos:prefLabel xml:lang="bg">Уницеф – Детски фонд на ООН</skos:prefLabel>
  <skos:prefLabel xml:lang="cs">UNICEF – Dětský fond Organizace spojených národů</skos:prefLabel>
  <skos:prefLabel xml:lang="da">UNICEF – De Forenede Nationers Børnefond</skos:prefLabel>
  <skos:prefLabel xml:lang="de">Unicef – Kinderhilfswerk der Vereinten Nationen</skos:prefLabel>
  <skos:prefLabel xml:lang="el">Unicef – Τομείο των Ηνωμένων Εθνών για τα Παιδιά</skos:prefLabel>
  <skos:prefLabel xml:lang="en">Unicef – United Nations Children's Fund</skos:prefLabel>
  ...
<skos:Concept rdf:about="http://publications.europa.eu/resource/authority/corporate-body/UNO"
  at:deprecated="false">
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNRWA"/>
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNICEF"/>
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/ECOSOC"/>
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNESCO"/>
  <at:authority-code>UNO</at:authority-code>
  <at:op-code>UNO</at:op-code>
  <atold:op-code>UNO</atold:op-code>
  <dc:identifier>UNO</dc:identifier>
  <at:start.use>1951-01-01</at:start.use>
  <skos:prefLabel xml:lang="bg">Организация на обединените нации</skos:prefLabel>
  <skos:prefLabel xml:lang="cs">Organizace spojených národů</skos:prefLabel>
  <skos:prefLabel xml:lang="da">De Forenede Nationers Organisation</skos:prefLabel>
  <skos:prefLabel xml:lang="de">Vereinte Nationen</skos:prefLabel>
  <skos:prefLabel xml:lang="el">Οργανισμός Ηνωμένων Εθνών</skos:prefLabel>
  <skos:prefLabel xml:lang="en">United Nations Organisation</skos:prefLabel>
```



Linked data foundations

- URIs for naming things
- RDF for describing data
- SPARQL for querying linked data



Uniform Resource Identifier (URI)

- “A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.”
 - ISA’s 10 Rules for Persistent URIs

- A country, e.g. Belgium
 - <http://publications.europa.eu/resource/authority/country/BEL>



- An organisation, e.g. the Publications Office
 - <http://publications.europa.eu/resource/authority/corporate-body/PUBL>



- A dataset, e.g. Countries Named Authority List
 - <http://publications.europa.eu/resource/authority/country/>

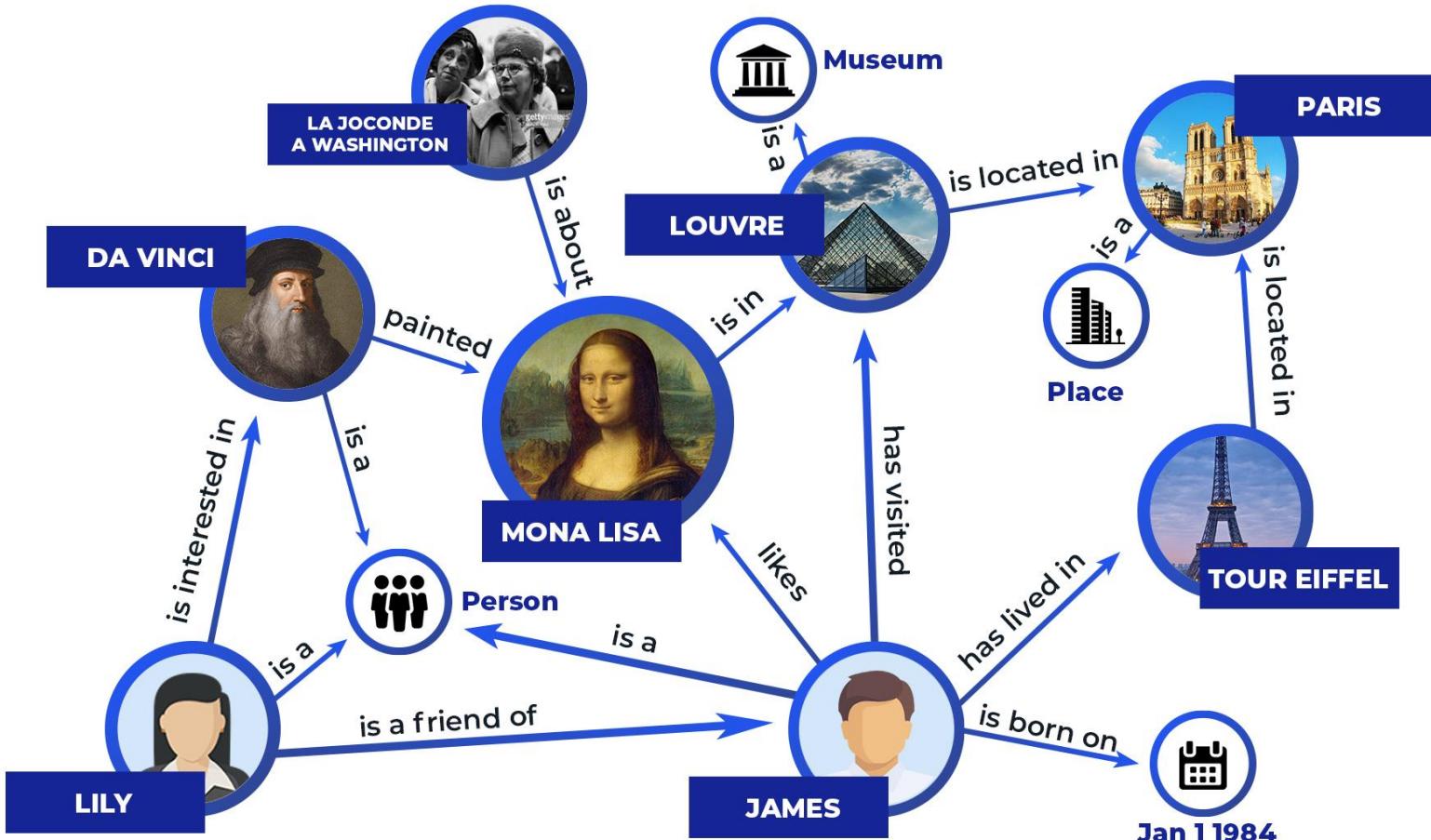




Outline

- Semi-Structured Data
 - XML
 - JSON
 - **RDF**
 - 4 Linked Open Data
 - 4 **RDF Structure**
 - 4 SPARQL
- Object Orientation
- Textual Data
- Spatial Data

Knowledge Graph





RDF

- The Resource Description Framework (RDF) is a syntax for representing data and resources in the Web
- RDF breaks every piece of information down in triples:
 - Subject – a resource, which may be identified with a URI.
 - Predicate – a URI-identified reused specification of the relationship.
 - Object – a resource or literal to which the subject is related.

<http://example.org/place/Brussels> is the capital of “Belgium”.

OR

<http://example.org/place/Brussels> is the capital of <http://example.org/place/Belgium>.

Subject

Predicate

Object



RDF Syntax (RDF/XML)

```
<rdf:RDF  
  
    xmlns:dcat="http://www.w3.org/TR/vocab-dcat/"  
    xmlns:dct="http://purl.org/dc/terms/"  
  
<dcat:Dataset rdf:about="http://publications.europa.eu/resource/authority/file-type/">  
    <dct:title> "File types Name Authority List" </dct:title>  
    <dct:publisher rdf:resource="http://open-data.europa.eu/en/data/publisher/publ"/>  
</dcat:Dataset>  
  
<dct:Agent rdf:about="http://open-data.europa.eu/en/data/publisher/publ"/>  
    <dct:title> "Publications Office" </dct:title>  
</dct:Agent>  
  
</rdf:RDF>
```

Graph

	Subject
	Predicate
	Object

RDF/XML is currently the only syntax that is standardised by W3C.



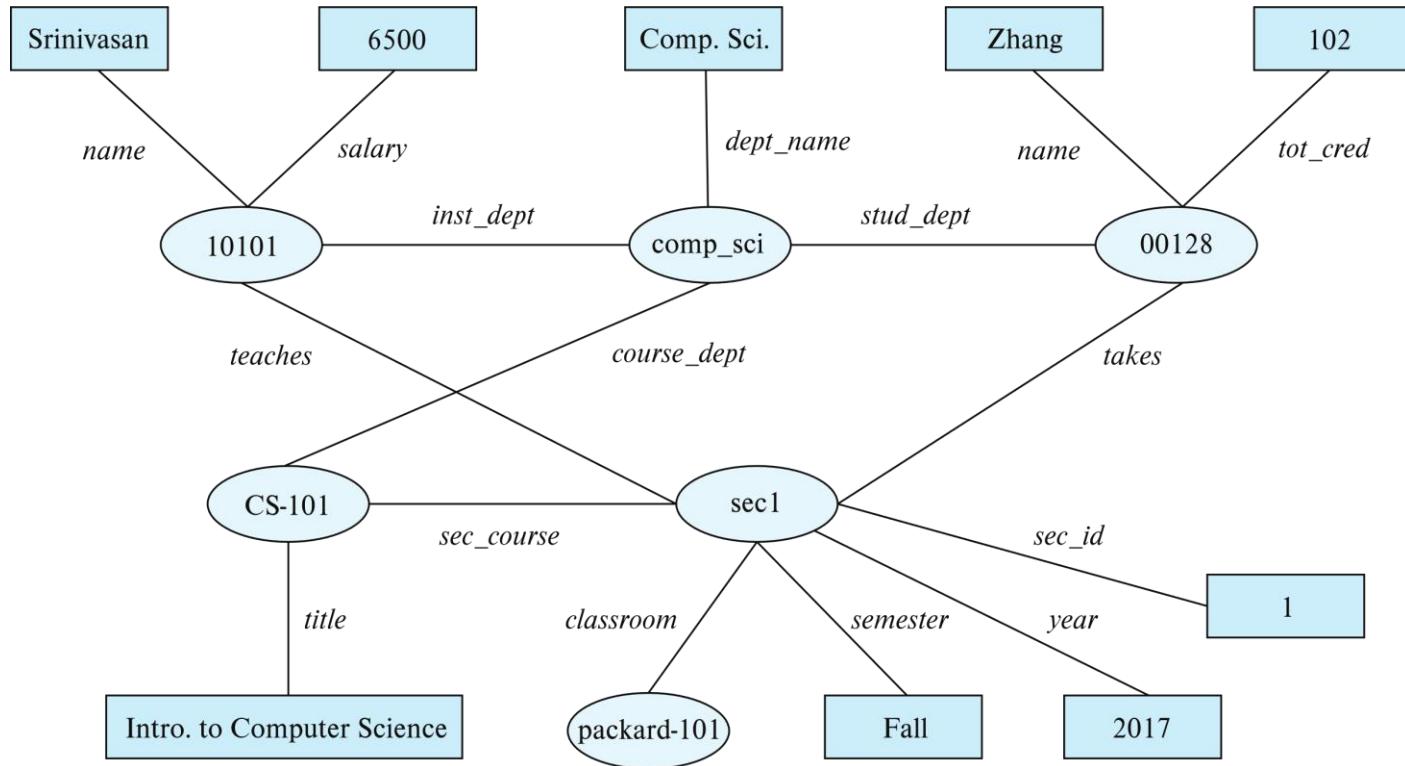
Knowledge Representation

- Representation of human knowledge is a long-standing goal of AI
 - Various representations of facts and inference rules proposed over time
- **RDF: Resource Description Format**
 - Simplified representation for facts, represented as triples (*subject, predicate, object*)
 - E.g., (NBA-2019, *winner*, Raptors)
(Washington-DC, *capital-of*, USA)
(Washington-DC, *population*, 6,200,000)
 - Models objects that have attributes, and relationships with other objects
 - Like the ER model, but with a flexible schema
 - (*ID, attribute-name, value*)
 - (*ID1, relationship-name, ID2*)
 - Has a natural graph representation



Graph View of RDF Data

- **Knowledge graph**





Triple View of RDF Data

10101	instance-of	instructor .
10101	name	"Srinivasan" .
10101	salary	"6500" .
00128	instance-of	student .
00128	name	"Zhang" .
00128	tot_cred	"102" .
comp_sci	instance-of	department .
comp_sci	dept_name	"Comp. Sci." .
biology	instance-of	department .
CS-101	instance-of	course .
CS-101	title	"Intro. to Computer Science" .
CS-101	course_dept	comp_sci .
sec1	instance-of	section .
sec1	sec_course	CS-101 .
sec1	sec_id	"1" .
sec1	semester	"Fall" .
sec1	year	"2017" .
sec1	classroom	packard-101 .
sec1	time_slot_id	"H" .
10101	inst_dept	comp_sci .
00128	stud_dept	comp_sci .
00128	takes	sec1 .
10101	teaches	sec1 .



RDF Representation

- RDF triples represent binary relationships
- How to represent n-ary relationships?
 - Approach 1 (from Section 6.9.4): Create artificial entity, and link to each of the n entities
 - E.g., (Barack Obama, *president-of*, USA, 2008-2016) can be represented as
(e1, *person*, Barack Obama), (e1, *country*, USA),
(e1, *president-from*, 2008) (e1, *president-till*, 2016)
 - Approach 2: use **quads** instead of triples, with context entity
 - E.g., (Barack Obama, *president-of*, USA, c1)
(c1, *president-from*, 2008) (c1, *president-till*, 2016)



RDF Representation

- RDF widely used as knowledge base representation
 - DBpedia, Yago, Freebase, WikiData, ..
- **Linked open data** project aims to connect different knowledge graphs to allow queries to span databases



Outline

- Semi-Structured Data
 - XML
 - JSON
 - **RDF**
 - 4 Linked Open Data
 - 4 RDF Structure
 - 4 **SPARQL**
- Object Orientation
- Textual Data
- Spatial Data



SPARQL

- SPARQL is a standardised language for querying RDF data.



Querying RDF: SPARQL

- Triple patterns
 - `?cid title "Intro. to Computer Science"`
`?sid course ?cid`
- SPARQL queries
 - **select** `?name`
where {
 `?cid title "Intro. to Computer Science" .`
 `?sid course ?cid .`
 `?id takes ?sid .`
 `?id name ?name .`
}
- Also supports
 - Aggregation, Optional joins (similar to outerjoins), Subqueries, etc.
 - Transitive closure on paths



Outline

- Semi-Structured Data
- **Object Orientation**
 - Complex Data Types and Object Orientation
 - Structured Data Types and Inheritance in SQL
 - Array and Multiset Types in SQL
 - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data



Object Orientation

- **Object-relational data model** provides richer type system
 - with complex data types and object orientation
- Applications are often written in object-oriented programming languages
 - Type system does not match relational type system
 - Switching between imperative language and SQL is troublesome



Object Orientation

- Approaches for integrating object-orientation with databases
 - Build an **object-relational database**, adding object-oriented features to a relational database
 - Automatically convert data between programming language model and relational model; data conversion specified by **object-relational mapping**
 - Build an **object-oriented database** that natively supports object-oriented data and direct access from programming language



Outline

- Semi-Structured Data
- **Object Orientation**
 - **Complex Data Types and Object Orientation**
 - Structured Data Types and Inheritance in SQL
 - Array and Multiset Types in SQL
 - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data



Outline

- Semi-Structured Data
- **Object Orientation**
 - **Complex Data Types and Object Orientation**
 - Structured Data Types and Inheritance in SQL
 - Array and Multiset Types in SQL
 - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data



Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.



Complex Data Types

- Motivation:
 - Permit non-atomic domains
 - Example of non-atomic domain: set of integers, or set of tuples
 - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
 - allow relations whenever we allow atomic (scalar) values — relations within relations
 - Retains mathematical foundation of relational model
 - Violates first normal form.



Example of a Nested Relation

- Example: library information system
- Each book has
 - title,
 - a list (array) of authors,
 - Publisher, with subfields *name* and *branch*, and
 - a set of keywords
- Non-1NF relation *books*

<i>title</i>	<i>author_array</i>	<i>publisher</i>	<i>keyword_set</i>
		(<i>name</i> , <i>branch</i>)	
Compilers	[Smith, Jones]	(McGraw-Hill, New York)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}



4NF Decomposition of Nested Relation

- Suppose for simplicity that title uniquely identifies a book
 - In real world ISBN is a unique identifier
- Decompose books into 4NF using the schemas:
 - (title, author, position)
 - (title, keyword)
 - (title, pub-name, pub-branch)
- 4NF design requires users to include joins in their queries.

title	author	position
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	keyword
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	pub_name	pub_branch
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4



Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
 - Collection and large object types
 - Nested relations are an example of collection types
 - Structured types
 - Nested record structures like composite attributes
 - Inheritance
 - Object orientation
 - Including object identifiers and references
- Not fully implemented in any database system currently
 - But some features are present in each of the major commercial database systems
 - Read the manual of your database system to see what it supports



Outline

- Semi-Structured Data
- **Object Orientation**
 - Complex Data Types and Object Orientation
 - **Structured Data Types and Inheritance in SQL**
 - Array and Multiset Types in SQL
 - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data



Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

```
create type Name as
  (firstname      varchar(20),
   lastname       varchar(20))
final
```

```
create type Address as
  (street        varchar(20),
   city          varchar(20),
   zipcode       varchar(20))
not final
```

- Note: **final** and **not final** indicate whether subtypes can be created



Structured Types and Inheritance in SQL

- Structured types can be used to create tables with composite attributes

```
create table person (
    name    Name,
    address Address,
    dateOfBirth date)
```

- Dot notation used to reference components: *name.firstname*



Structured Types (cont.)

- **User-defined row types**

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
not final
```

- Can then create a table whose rows are a user-defined type
create table person of PersonType

- Alternative using **unnamed row types**.

```
create table person_r(
    name    row(firstname varchar(20),
                lastname  varchar(20)),
    address row(street    varchar(20),
                city      varchar(20),
                zipcode   varchar(20)),
    dateOfBirth date)
```



Methods

- Can add a method declaration with a structured type.

method ageOnDate (onDate date)

returns interval year

- Method body is given separately.

create instance method ageOnDate (onDate date)

returns interval year

for PersonType

begin

return onDate - self.dateOfBirth;

end

- We can now find the age of each person:

select name.lastname, ageOnDate (current_date)

from person



Constructor Functions

- **Constructor functions** are used to create values of structured types

- E.g.

```
create function Name(firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

- To create a value of type *Name*, we use

```
new Name('John', 'Smith')
```

- Normally used in insert statements

```
insert into Person values
```

```
(new Name('John', 'Smith),
  new Address('20 Main St', 'New York', '11001'),
  date '1960-8-22');
```



Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person  
(name varchar(20),  
address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student  
under Person  
(degree varchar(20),  
department varchar(20))
```

```
create type Teacher  
under Person  
(salary integer,  
department varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration



Multiple Type Inheritance

- SQL:1999 and SQL:2003 do not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

```
create type Teaching Assistant  
under Student, Teacher
```

- To avoid a conflict between the two occurrences of *department* we can rename them

```
create type Teaching Assistant  
under  
Student with (department as student_dept),  
Teacher with (department as teacher_dept)
```

- Each value must have a **most-specific type**



Table Inheritance

- Tables created from subtypes can further be specified as **subtables**
- E.g.

```
create table people of Person;  
create table students of Student under people;  
create table teachers of Teacher under people;
```
- Tuples added to a subtable are automatically visible to queries on the supertable
 - E.g. query on *people* also sees *students* and *teachers*.
 - Similarly updates/deletes on *people* also result in updates/deletes on subtables
 - To override this behaviour, use “**only people**” in query



Outline

- Semi-Structured Data
- **Object Orientation**
 - Complex Data Types and Object Orientation
 - Structured Data Types and Inheritance in SQL
 - **Array and Multiset Types in SQL**
 - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data



Array and Multiset Types in SQL

- SQL supports two collection types:
 - Arrays
 - array types were added in SQL:1999
 - Multisets
 - multiset types were added in SQL:2003 .
- Multiset is an unordered collection, where an element may occur multiple times.
- Multisets are like sets, except that a set allows each element to occur at most once.
- Unlike elements in a multiset, the elements of an array are ordered
 - So we can distinguish the first item from the second item, and so on.



Array and Multiset Types in SQL

- Example of array and multiset declaration:

```
create type Publisher as  
  (name      varchar(20),  
   branch    varchar(20));
```

```
create type Book as  
  (title      varchar(20),  
   author_array varchar(20) array [10],  
   pub_date   date,  
   publisher  Publisher,  
   keyword-set varchar(20) multiset);
```

```
create table books of Book;
```



Creation of Collection Values

- Array construction

```
array ['Silberschatz', `Korth', `Sudarshan']
```

- Multisets

```
multiset ['computer', 'database', 'SQL']
```

- To create a tuple of the type defined by the books relation:

```
('Compilers', array[` Smith', ` Jones'],
  new Publisher(` McGraw-Hill', ` New York'),
  multiset [` parsing', ` analysis' ])
```

- To insert the preceding tuple into the relation books

```
insert into books
```

```
values
```

```
('Compilers', array[` Smith', ` Jones'],
  new Publisher(` McGraw-Hill', ` New York'),
  multiset [` parsing', ` analysis' ]);
```



Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,

```
select title  
from books  
where 'database' in (unnest(keyword-set))
```

- We can access individual elements of an array by using indices
 - E.g.: If we know that a particular book has three authors, we could write:

```
select author_array[1], author_array[2], author_array[3]  
from books  
where title = `Database System Concepts`
```

- To get a relation containing pairs of the form “title, author_name” for each book and each author of the book

```
select B.title, A.author  
from books as B,  
unnest (B.author_array) as A (author )
```



Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**.

- E.g.

```
select title, A as author, publisher.name as pub_name,  
       publisher.branch as pub_branch, K.keyword  
from books as B,  
     unnest(B.author_array ) as A (author ),  
     unnest (B.keyword_set ) as K (keyword )
```

- Result relation *flat_books*

<i>title</i>	<i>author</i>	<i>pub_name</i>	<i>pub_branch</i>	<i>keyword</i>
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web



Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- Nesting can be done in a manner similar to aggregation, but using the function **collect()** in place of an aggregation operation, to create a multiset
- To nest the *flat_books* relation on the attribute *keyword*:

```
select title, author, Publisher (pub_name, pub_branch ) as publisher,  
      collect (keyword) as keyword_set  
from flat_books  
groupby title, author, publisher
```

<i>title</i>	<i>author</i>	<i>publisher</i> (<i>pub_name, pub_branch</i>)	<i>keyword_set</i>
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}



Nesting

- To nest on both authors and keywords:

```
select title,  
      collect (author ) as author_set,  
      Publisher (pub_name, pub_branch) as publisher,  
      collect (keyword ) as keyword_set  
  from flat_books  
 group by title, publisher
```



Nesting

- To nest on both authors and keywords using set():

```
select title,  
      set (author ) as author_set,  
      Publisher (pub_name, pub_branch) as publisher,  
      set (keyword ) as keyword_set  
  from flat_books  
 group by title, publisher
```



Nesting

- To nest on both authors and keywords using subqueries:

```
select title,  
       ( select author  
         from flat-books as M  
         where M.title=O.title) as author-set,  
       Publisher (pub-name, pub-branch) as publisher,  
       ( select keyword  
         from flat-books as N  
         where N.title = O.title ) as keyword-set  
     from flat-books as O
```



Nesting

- To nest on both authors and keywords using subqueries:

```
select title,  
       array( select author  
              from authors as A  
             where A.title = B.title  
             order by A.position) as author_array,  
       Publisher(pub_name, pub_branch) as publisher,  
       multiset( select keyword  
                  from keywords as K  
                 where K.title = B.title) as keyword_set,  
       from books4 as B;
```



Outline

- Semi-Structured Data
- **Object Orientation**
 - Complex Data Types and Object Orientation
 - Structured Data Types and Inheritance in SQL
 - Array and Multiset Types in SQL
 - **Comparison of Object-Oriented and Object-Relational Databases**
- Textual Data
- Spatial Data



Object-Relational Mapping

- **Object-Relational Mapping (ORM)** systems built on top of traditional relational databases
- Implementer provides a mapping from objects to relations
- Objects can be retrieved from database
 - System uses mapping to fetch relevant data from relations and construct objects
 - Updated objects are stored back in database by generating corresponding update/insert/delete statements



Object-Relational Mapping

- Object-relational mapping (ORM) systems allow
 - Specification of mapping between programming language objects and database tuples
 - Automatic creation of database tuples upon creation of objects
 - Automatic update/delete of database tuples when objects are update/deleted
 - Interface to retrieve objects satisfying specified conditions
 - Tuples in database are queried, and object created from the tuples
- Details in Section 9.6.2
 - Hibernate ORM for Java
 - Django ORM for Python



Comparison of O-R Databases

- Object-relational databases are object-oriented databases built on top of the relation model
- Object-relational mapping systems build an object layer on top of a traditional relational database
- Another type:
 - Object-oriented databases
- Each of these approaches targets a different market.



Comparison of O-R Databases

- Object-relational systems aim at making data modeling and querying easier by using complex data types.
 - Typical applications include storage and querying of complex data, including multimedia data.
 - SQL , however, imposes a significant performance penalty for certain kinds of applications that run primarily in main memory, and that perform a large number of accesses to the database.



Comparison of O-R Databases

- Object-relational mapping systems allow programmers to build applications using an object model, while using a traditional database system to store the data.
 - They combine the robustness of widely used relational database systems, with the power of object models for writing applications.
 - However, they suffer from overheads of data conversion between the object model and the relational model used to store data.



Comparison of O-R Databases

- **Relational systems**
 - simple data types, powerful query languages, high protection.
- **Object-relational systems**
 - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
 - complex data types integrated with programming language, but built as a layer on top of a relational database system



Outline

- Semi-Structured Data
- Object Orientation
- **Textual Data**
 - TF-IDF
 - Link Analysis
 - Evaluation
- Spatial Data

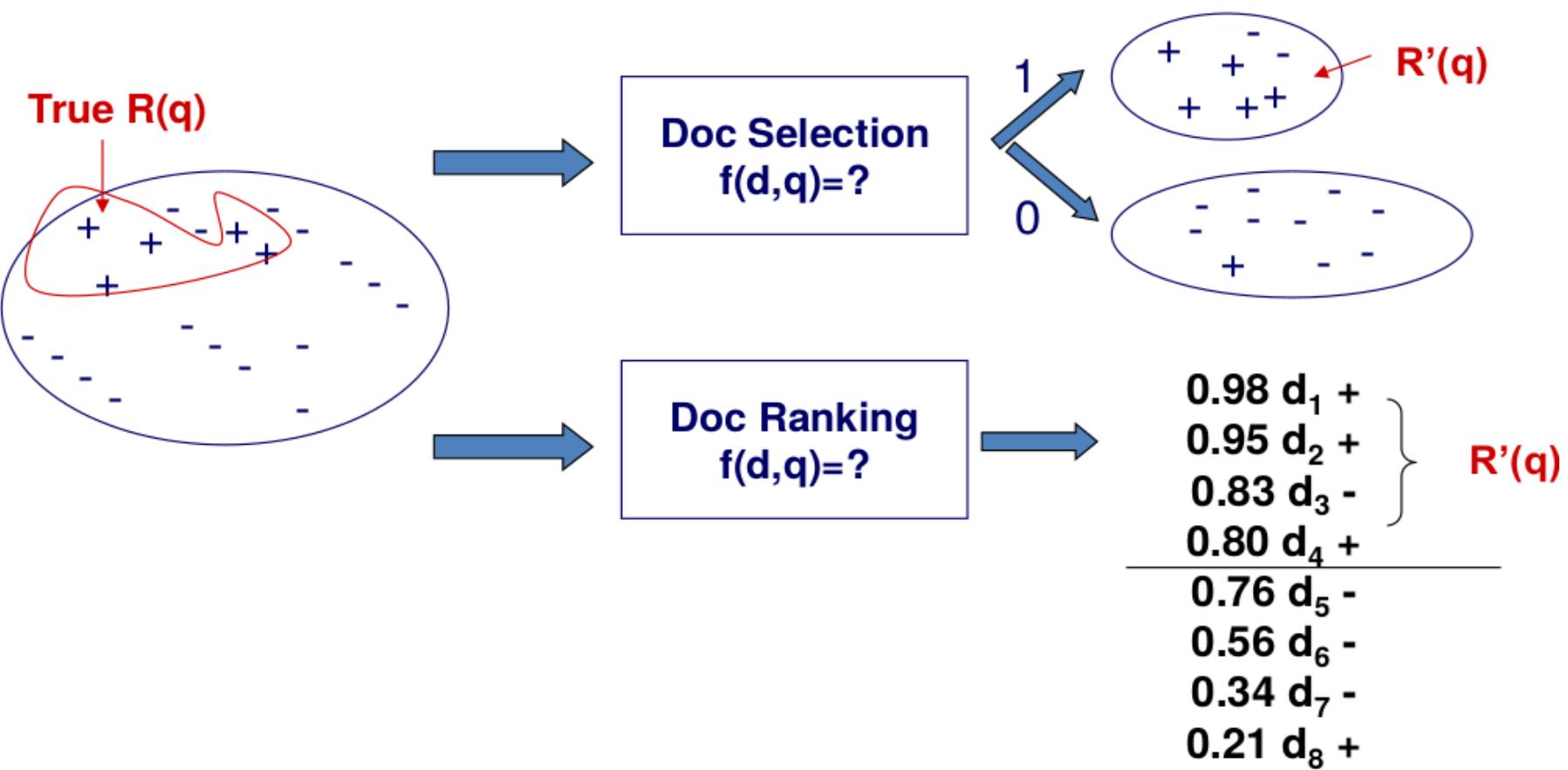


Textual Data

- **Information retrieval:** querying of unstructured data
 - Simple model of keyword queries: given query keywords, retrieve documents containing all the keywords
 - Today, keyword queries return many types of information as answers
 - E.g., a query “cricket” typically returns scores from ongoing or recent cricket matches, rather than just top-ranked documents related to cricket
- More advanced models rank relevance of documents
- Relevance ranking is essential since there are usually many documents matching keywords



Document Selection vs Ranking





Document Selection Problem

- The classifier is unlikely accurate
 - “Over-constrained” query no relevant documents to return
 - “Under-constrained” query over delivery
 - Hard to find the right position between these two extremes
- Even if it is accurate, all relevant documents are not equally relevant (relevance is a matter of degree!)
 - Prioritization is needed
- Thus, ranking is generally preferred => Vector Space Model



Ranked Retrieval

- Providing a relevance ranking of the documents with respect to a query
 - Assign a score to each query-document pair, say in [0, 1].
 - This score measures how well document and query “match”.
 - Sort documents according to scores



Vector Space Model

- Represent a doc/query by a term vector
 - Term: basic concept, e.g., word
 - Each term defines one dimension
 - N terms define an N-dimensional space
 - Query vector: $q=(x_1, \dots, x_N)$, x_i is query term weight
 - Doc vector: $d=(y_1, \dots, y_N)$, y_j is doc term weight
- $\text{relevance}(q,d) \approx \text{similarity}(q,d) = f(q,d)$



Vector Space Model

- Main items in calculating scores

- The importance of the term in query and document:
 - how many times does a query term occur in q and d?
=> Term Frequency (TF)
- The general importance of the term in the collection:
 - is it a frequent or rare term? how often do we see the query term in the entire collection?
=> Document Frequency (DF)
- Normalization of the scores based on the length of the document:
 - How long is d?
=> Document length ($|d|$)



Ranking using TF-IDF

- Term: keyword occurring in a document/query
- **Term Frequency:** $TF(d, t)$, the relevance of a term t to a document d
 - One definition: $TF(d, t) = \log(1 + n(d,t)/n(d))$
where
 - $n(d,t)$ = number of occurrences of term t in document d
 - $n(d)$ = number of terms in document d

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
ANTHONY	157	73	0	0	0	1	
BRUTUS	4	157	0	2	0	0	
CAESAR	232	227	0	2	1	0	
CALPURNIA	0	10	0	0	0	0	
CLEOPATRA	57	0	0	0	0	0	
MERCY	2	0	3	8	5	8	
WORSER	2	0	1	1	1	5	
...



Ranking using TF-IDF

- **Inverse document frequency:** $IDF(t)$
 - One definition: $IDF(t) = 1/n(t)$
- **Relevance** of a document d to a set of terms Q
 - One definition: $r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$
 - Other definitions
 - take **proximity** of words into account
 - **Stop words** are often ignored

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0



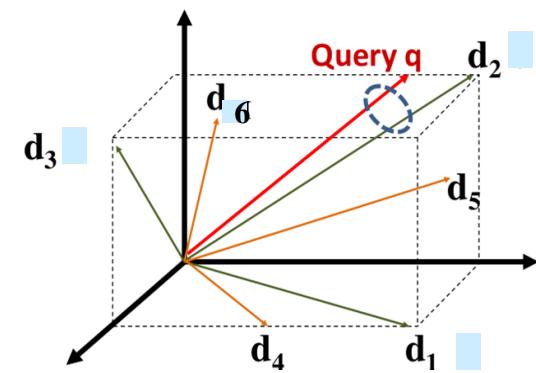
Document Normalization

- Long doc has a better chance to match any query
- Penalize a long documents with a doc length normalizer

Cosine Similarity

$$\cos(\vec{q}, \vec{d}) = \text{SIM}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| |\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- cosine similarity of \vec{q} and \vec{d}
 - q_i is the *tf-idf* weight of term i in the query.
 - d_i is the *tf-idf* weight of term i in the document.
 - $|\vec{q}|$ and $|\vec{d}|$ are the lengths of \vec{q} and \vec{d}
- Also includes doc length normalization





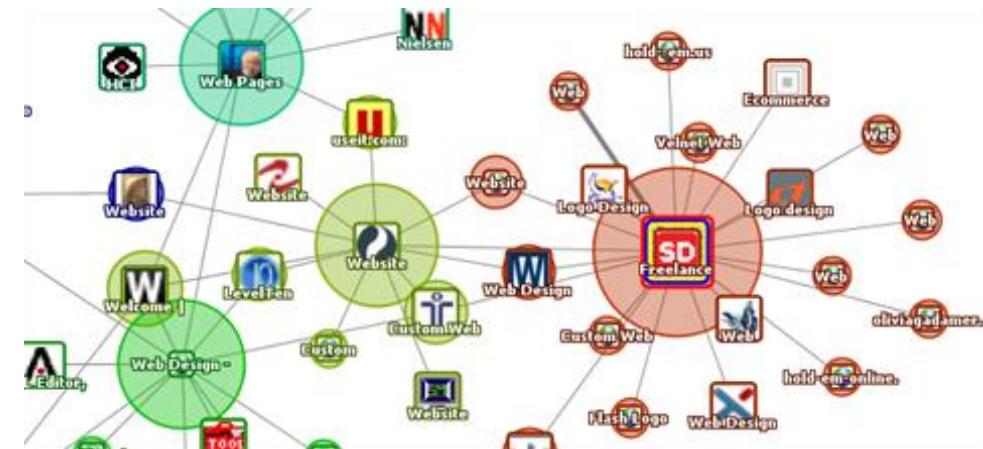
Outline

- Semi-Structured Data
- Object Orientation
- **Textual Data**
 - TF-IDF
 - **Link Analysis**
 - Evaluation
- Spatial Data

Link Analysis



- Makes use of links and anchor text in web pages.
 - Stored and indexed separately
 - <a href = <http://www.aut.ac.ir>> Amirkabir University of Technology
- Link analysis identifies popularity and community information
 - e.g., PageRank
- Anchor text can significantly enhance the representation of pages pointed to by links
- Significant impact on web search
 - Less importance in other applications





Ranking Using Hyperlinks

- Hyperlinks provide very important clues to importance
- Google introduced PageRank, a measure of popularity/importance based on hyperlinks to pages
 - Pages hyperlinked from many pages should have higher PageRank
 - Pages hyperlinked from pages with higher PageRank should have higher PageRank
 - Formalized by **random walk** model



Ranking Using Hyperlinks

- Let $T[i, j]$ be the probability that a random walker who is on page i will click on the link to page j
 - Assuming all links are equal, $T[i, j] = 1/N_i$
- Then PageRank[j] for each page j can be defined as
 - $P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^N (T[i, j] * P[i])$
 - N = total number of pages,
 - δ a constant usually set to 0.15



Ranking Using Hyperlinks

- Definition of PageRank is circular, but can be solved as a set of linear equations
 - Simple iterative technique works well
 - Initialize all $P[i] = 1/N$
 - In each iteration use equation $P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^N (T[i, j] * P[i])$ to update P
 - Stop iteration when changes are small, or some limit (say 30 iterations) is reached.



Other Aspects

- Other measures of relevance are also important. For example:
 - Keywords in anchor text
 - Number of times who ask a query click on a link if it is returned as an answer



Outline

- Semi-Structured Data
- Object Orientation
- **Textual Data**
 - TF-IDF
 - Link Analysis
 - **Evaluation**
- Spatial Data

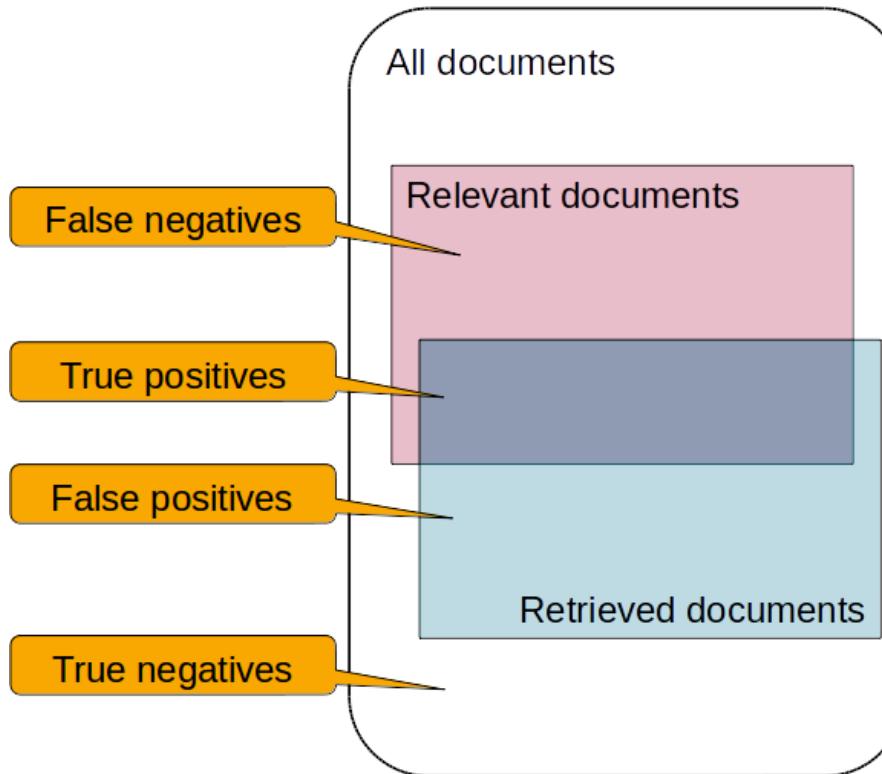


Precision and Recall

- Required data
 - The set of relevant documents (but we may not find all)
 - The set of retrieved documents (but not all of them are relevant)
- Precision (P) is the fraction of retrieved documents that are relevant
$$\text{Precision} = \#(\text{relevant items retrieved}) / \#(\text{retrieved items}) = P(\text{relevant}|\text{retrieved})$$
- Recall (R) is the fraction of relevant documents that are retrieved
$$\text{Recall} = \#(\text{relevant items retrieved}) / \#(\text{relevant items}) = P(\text{retrieved}|\text{relevant})$$



Precision and Recall



Precision =
True positives
Retrieved documents

Recall =
True positives
Relevant documents



Precision/Recall Tradeoff

- Find algorithm that maximizes precision.
 - Or minimizes classification errors (false positives)
 - Return nothing!
- Find algorithm that maximizes recall.
 - Return everything!
- Solution:
 - Considering both measures at the same time by F-Measure



F-Measure

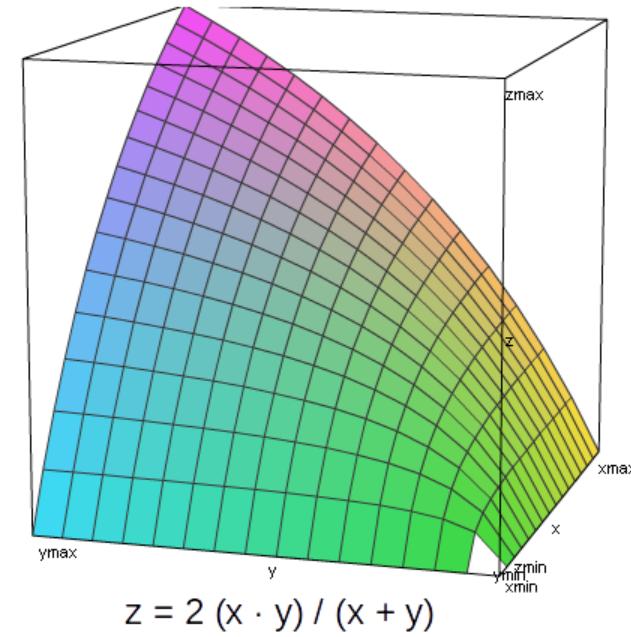
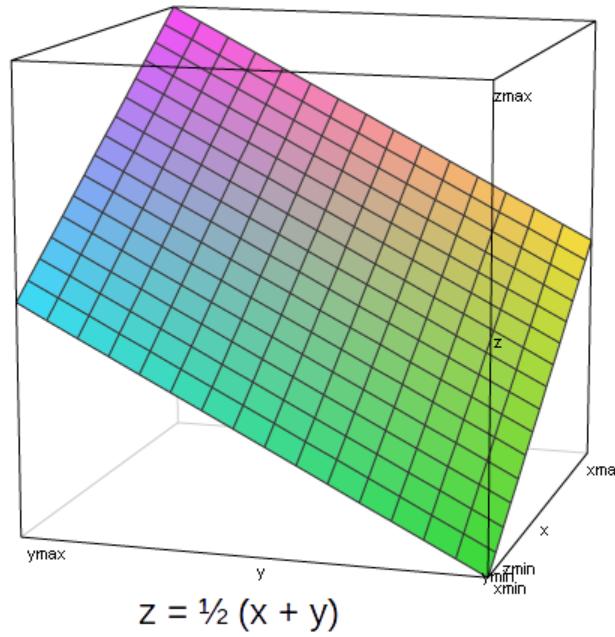
- General formula

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1 - \alpha}{\alpha}$$

$\alpha \in [0, 1]$ and thus $\beta^2 \in [0, \infty]$

- Harmonic mean of recall and precision ($\beta^2 = 1$)

$$F = \frac{1}{\frac{1}{2} \left(\frac{1}{R} + \frac{1}{P} \right)} = \frac{2RP}{R + P}$$





Retrieval Effectiveness

- Measures of effectiveness
 - **Precision:** what percentage of returned results are actually relevant
 - **Recall:** what percentage of relevant results were returned
 - At some number of answers, e.g. precision@10, recall@10



Retrieval Effectiveness

- Measures of effectiveness
 - **Precision:** what percentage of returned results are actually relevant
 - **Recall:** what percentage of relevant results were returned
 - At some number of answers, e.g. precision@10, recall@10



Outline

- Semi-Structured Data
- Object Orientation
- Textual Data
- **Spatial Data**



Spatial Data

- Spatial databases store information related to spatial locations, and support efficient storage, indexing and querying of spatial data.
 - **Geographic data** -- road maps, land-usage maps, topographic elevation maps, political maps showing boundaries, land-ownership maps, and so on.
 - **Geographic information systems** are special-purpose databases tailored for storing geographic data.
 - Round-earth coordinate system may be used
 - (Latitude, longitude, elevation)
 - **Geometric data:** design information about how objects are constructed .
 - For example, designs of buildings, aircraft, layouts of integrated-circuits.
 - 2 or 3 dimensional Euclidean space with (X, Y, Z) coordinates



Represented of Geometric Information

Various geometric constructs can be represented in a database in a normalized fashion

- A **line segment** can be represented by the coordinates of its endpoints.
- A **polyline** or **linestring** consists of a connected sequence of line segments and can be represented by a list containing the coordinates of the endpoints of the segments, in sequence.
 - Approximate a curve by partitioning it into a sequence of segments
 - Useful for two-dimensional features such as roads.
 - Some systems also support *circular arcs* as primitives, allowing curves to be represented as sequences of arc
- **Polygons** is represented by a list of vertices in order.
 - The list of vertices specifies the boundary of a polygonal region.
 - Can also be represented as a set of triangles (**triangulation**)



Representation of Geometric Constructs

object	representation
line segment	 1 → 2 $\{(x_1, y_1), (x_2, y_2)\}$
triangle	 1 → 2 → 3 → 1 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$
polygon	 1 → 2 → 3 → 4 → 5 → 1 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5)\}$
polygon	 1 → 2 → 3 → 4 → 5 → 1 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), ID1\}$ $\{(x_1, y_1), (x_3, y_3), (x_4, y_4), ID1\}$ $\{(x_1, y_1), (x_4, y_4), (x_5, y_5), ID1\}$



Representation of Geometric Information (Cont.)

- Representation of points and line segment in 3-D similar to 2-D, except that points have an extra z component
- Represent arbitrary polyhedra by dividing them into tetrahedrons, like triangulating polygons.
- Alternative: List their faces, each of which is a polygon, along with an indication of which side of the face is inside the polyhedron.
- Geometry and geography data types supported by many databases
 - E.g. SQL Server and PostGIS
 - point, linestring, curve, polygons
 - Collections: multipoint, multilinestring, multicurve, multipolygon
 - `LINESTRING(1 1, 2 3, 4 4)`
 - `POLYGON((1 1, 2 3, 4 4, 1 1))`
 - Type conversions:
 - `ST_GeometryFromText()` and `ST_GeographyFromText()`
 - Operations:
 - `ST_Union()`, `ST_Intersection()`, ...



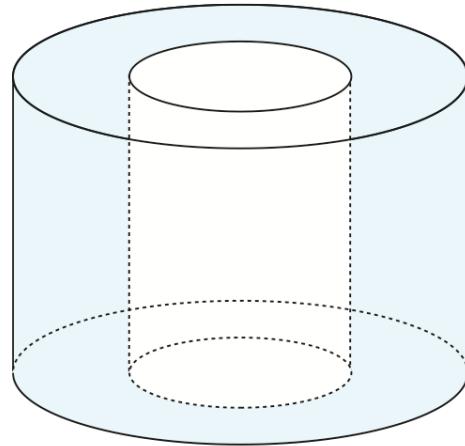
Design Databases

- Represent design components as objects (generally geometric objects); the connections between the objects indicate how the design is structured.
- Simple two-dimensional objects: points, lines, triangles, rectangles, polygons.
- Complex two-dimensional objects: formed from simple objects via union, intersection, and difference operations.
- Complex three-dimensional objects: formed from simpler objects such as spheres, cylinders, and cuboids, by union, intersection, and difference operations.
- Wireframe models represent three-dimensional surfaces as a set of simpler objects.

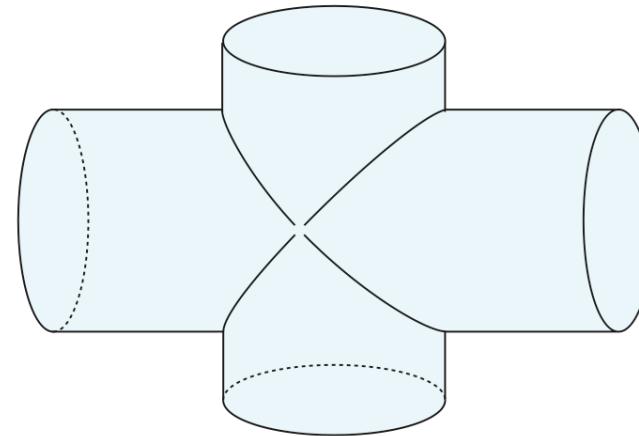


Representation of Geometric Constructs

- Design databases also store non-spatial information about objects (e.g., construction material, color, etc.)
- Spatial integrity constraints are important.
 - E.g., pipes should not intersect, wires should not be too close to each other, etc.



(a) Difference of cylinders



(b) Union of cylinders



Geographic Data

- **Raster data** consist of bit maps or pixel maps, in two or more dimensions.
 - Example 2-D raster image: satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area.
 - Additional dimensions might include the temperature at different altitudes at different regions, or measurements taken at different points in time.
- Design databases generally do not store raster data.



Geographic Data (Cont.)

- **Vector data** are constructed from basic geometric objects: points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions.
- Vector format often used to represent map data.
 - Roads can be considered as two-dimensional and represented by lines and curves.
 - Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant.
 - Features such as regions and lakes can be depicted as polygons.



Spatial Queries

- **Region queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region
 - E.g., PostGIS *ST_Contains()*, *ST_Overlaps()*, ...
- **Nearness queries** request objects that lie near a specified location.
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Spatial graph queries** request information based on spatial graphs
 - E.g., shortest path between two points via a road network
- **Spatial join** of two spatial relations with the location playing the role of join attribute.
- Queries that compute intersections or **unions** of regions



Questions?